THE UNIVERSITY OF CHICAGO

LEVERAGING SYMMETRY AND STRUCTURE IN MACHINE LEARNING

A DISSERTATION SUBMITTED TO
THE FACULTY OF THE DIVISION OF THE PHYSICAL SCIENCES
IN CANDIDACY FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

DEPARTMENT OF COMPUTER SCIENCE

BY

HORACE PAN

CHICAGO, ILLINOIS
AUGUST 2022

For my parents

TABLE OF CONTENTS

LIST OF FIGURES vii

LIST OF TABLES ix

ACKNOWLEDGMENTS x

ABSTRACT xi

I PERMUTATION EQUIVARIANT LAYERS FOR HIGHER ORDER INTERACTIONS

1 INTRODUCTION 3
- 1.1 Preliminaries/Notation 5
  - 1.1.1 Groups 5
  - 1.1.2 Neural Networks 9
  - 1.1.3 Permutation Equivariance 10
- 1.2 Problem Statement 13

2 RELATED WORK 14
- 2.1 Early Permutation Invariant Architectures 14
  - 2.1.1 PointNet 14
  - 2.1.2 Deep Sets 16
- 2.2 Parameter Sharing and Permutation Equivariance 20
- 2.3 Graph Neural Networks 23
  - 2.3.1 Message Passing Neural Networks 25
  - 2.3.2 Convolutional Networks on Graphs for Learning Molecular Fingerprints 27
  - 2.3.3 Gated Graph Sequence Networks 27
  - 2.3.4 GraphSage 28
  - 2.3.5 Graph Attention Networks 29
  - 2.3.6 Covariant Composition Networks 29
  - 2.3.7 Invariant and Equivariant Graph Networks 32
- 2.4 Attention 34
  - 2.4.1 Self Attention 36

3 HIGHER ORDER PERMUTATION EQUIVARIANT LAYERS 38
- 3.1 Deriving Equivariant Layers 38
  - 3.1.1 Case I: 1 to 1 layer 38
  - 3.1.2 Case II: 1 to 2 layer 39
  - 3.1.3 Case III: 2 to 1 layer 40
  - 3.1.4 Case IV: 2 to 2 Layer 40
- 3.2 Deriving Equivariant Layers From Partitions 41
- 3.3 Organizing the computation 44

3.3.1 Summation Tensors . . . 44
3.3.2 Transfer Operations . . . 45
3.3.3 Broadcast Tensors . . . 46
3.3.4 Constructing a Broadcast Tensor for a Specific Partition . . . 48
3.3.5 Making the Layer Learnable . . . 49
3.4 EQUIVARIANT LAYER DETAILS . . . 50
3.4.1 Complexity of Operations . . . 50
3.4.2 Linearly Mixing the Broadcast Tensors . . . 51
3.5 Implementation . . . 52
3.5.1 Architecture . . . 53

4 EXPERIMENTS . . . 59
4.1 Predicting Poker Hands . . . 59
4.2 Counting Unique Elements of a Set . . . 61
4.3 Predicting the Efficacy of Drug Cocktails . . . 63
4.4 Jet Tagging . . . 64
4.5 Discussion . . . 66

5 CONCLUSION . . . 68
5.1 Concluding Remarks . . . 68
5.2 Future Work . . . 69

II FOURIER BASES FOR SOLVING PERMUTATION PUZZLES

6 INTRODUCTION . . . 73

7 PRELIMINARIES . . . 76
7.1 Groups . . . 76
7.2 Fourier Analysis on Finite Groups . . . 78

8 REINFORCEMENT LEARNING BACKGROUND . . . 82
8.1 Markov Decision Processes . . . 82
8.2 Value Iteration and Approximate Dynamic Programming . . . 84

9 RELATED WORK . . . 87
9.1 Proto-Value Functions . . . 87
9.2 Heuristic Search in Fourier Space . . . 88
9.3 Deep Value Networks . . . 90

10 LEARNING A VALUE FUNCTION IN THE FOURIER BASIS . . . 92
10.1 Low Rank Fourier Matrices . . . 93
10.2 Algorithm . . . 93

11 CONSTRUCTING THE FOURIER BASIS 95
11.1 Preliminaries 95
11.2 Defining Young's Orthogonal Representation on Transpositions 97
11.3 Dimensions of the Irreps of the Symmetric Group 99
11.4 Irreducible Representations of Wreath Product Groups 100
11.4.1 Irreducible Representations of C_m^n 100
11.4.2 Young Subgroups 101
11.4.3 Induced Representations 102
11.4.4 Putting it all together 103
11.5 Implementation Details for Computing Wreath Product Irreps 105

12 EXPERIMENTS 107
12.1 Baseline DVN 108
12.2 Hyperparameters 109
12.3 Evaluation 111
12.4 Heuristic search with the value function 112
12.5 Learning low rank Fourier models 113
12.6 Discussion 113
12.6.1 Miscellaneous Experiment Details 115
12.6.2 Hyperparameters 116
12.6.3 Irreps 116

13 CONCLUSION 119
13.1 Concluding Remarks 119
13.2 Future Work 119

REFERENCES 122

LIST OF FIGURES

1.1 Images exhibit rotational and translational symmetry. 3
1.2 Molecules (1.2a) and point clouds (1.2b) exhibit 3D rotational symmetry. Vertices of the graph and points of the point cloud have permutation symmetry. 4
1.3 Items in a set have no inherent ordering and can be permuted arbitrarily without affecting their composition. 5
1.4 Relabeling vertices v_1, v_2, v_3 with permutation $\pi = (1, 2, 3) \in \mathbb{S}_3$ induces the change in the graph and adjacency matrix (a 2nd order permutable tensor) above. 9

2.1 Figure from Qi et al. [2017]. Each point of the point cloud is input as a 3 dimensional vector that is subsequential encoded by nonlinear transformations. The collection of 1024-dimensional representations of the points is aggregated with a max pooling operation to generate a global feature vector for the point cloud. Finally, this global feature vector goes through a final multilayer perceptron to produce the output classification of the network. 16
2.2 On input $x_1, x_2, \ldots, x_n \in \mathbb{R}^d$, a Deep Sets permutation equivariant layer outputs a linear combination of the individual x_i's and their sum $\sum_j x_j$. 17
2.3 User-Item affinity data. Each entry of the matrix is a score for how much a user enjoys the given fruit. 21
2.4 Figure from Hartford et al. [2018]. The pink cube indicates an element in the output. It is a linear combination of the corresponding element in the input matrix (in dark blue), row sums (in green), column sums (in yellow), sums over the entire matrix (Pooling structure implied by the tied weights for matrices (left) and 3D tensors (right). The pink cube highlights one element of the output. It is calculated as a function of the corresponding element from the input (dark blue), pooled aggregations over the rows and columns of the input (green and yellow), and pooled aggregation over the whole input matrix (red). In the tensor case (right), we pool over all sub-tensors (orange and purple submatrices, green sub-vectors and red scalar). For clarity, the output connections are not shown in the tensor case 23
2.5 Each node v_i has an associated hidden representation $h_{v_i}^{(t)}$ during the tth round of message passing. In the graph above, vertex v_3 collects messages from its neighbors v_1, v_2 and v_4, to produce its updated node representation $h_{v_3}^{(t+1)}$. . . . 25
2.6 A node's hidden representation in a CCN [Kondor et al., 2018b] can be a higher order permutable tensor. In the figure above, each node passes a 2^{nd} order permutable tensor to its neighbors during the message passing step of a CCN. The depth of the visualized node representation tensors is the feature dimension. 30

3.1 Constructing the broadcast tensors for a $2 \to 2$ equivariant layer 55
3.2 Constructing a subset of the broadcast tensors for a $3 \to 3$ equivariant layer . . 56
3.3 Implementation of $2 \to 2$ equivariant layer 57
3.4 Implementation of $3 \to 3$ equivariant layer 57

3.5 Implementation of a 2nd order permutation equivariant network with two $2 \to 2$ equivariant layers. The encoder and decoders can also be multilayer perceptrons. 58
3.6 Implementation of a 3rd order permutation equivariant network with two $3 \to 3$ equivariant layers. 58

7.1 2-by-2 Cube: the three possible orientations of a given corner cubie. 78
7.2 Suppose the bottom face of the Pyraminx puzzle in Fig 7.2a is yellow. Rotating the puzzle's front two layers clockwise by $\frac{2\pi}{3}$ results in Fig 7.2b. Subsequently, rotating just the front tip counterclockwise by $\frac{2\pi}{3}$ results in Fig 7.2c. 79

9.1 The 8-puzzle, also known as the 3-by-3 sliding number tile puzzle. The puzzle starts in a scrambled state and the player tries to arrange the tiles back into sorted order by maneuvering tiles into the empty free slot. 89
9.2 Agostinelli et al. [2019]'s deep value network for solving the Rubik's Cube . . . 91

12.1 Deep Value Network architecture for Pyraminx, $\mathbb{S}_8$ and 2-by-2 cube puzzles. Every fully connected layer (besides the final layer) is followed by a rectified linear unit activation. This DVN architecture is inspired by the architecture used by McAleer et al. [2018] and Agostinelli et al. [2019]. 108
12.2 Proportion of locally optimal moves 111
12.3 Proportion of puzzles solved with greedy policy 111
12.4 Training curve of low rank models for 2-by-2 cube 115

LIST OF TABLES

3.1 Operations to construct the broadcast tensors in a 2 to 2 layer 41

4.1 Model results on poker hand prediction 61
4.2 Poker Model Hyperparameters 61
4.3 Test accuracy on unique characters task 62
4.4 Hyperparameters for unique characters experiments 62
4.5 Test MAE, RMSE on drug combination efficacy prediction task 64
4.6 Test accuracy on jets tagging dataset 66

7.1 Dimensionality of irreps of $C_3 \wr \mathbb{S}_8$ 81

11.1 Dimension of irreps of $\mathbb{S}_6$ 100

12.1 Solve statistics for one irrep Fourier models on 2-by-2 cube after $50,000$ training epochs. 110
12.2 Proportion of puzzles solved by greedy search 112
12.3 2-by-2 cube A^* search solve statistics 114
12.4 Hyperparameters 116
12.5 Top 24 irreps used for Pyraminx 117
12.6 Top 4 irreps used for $\mathbb{S}_8$ 117
12.7 Top 2 irreps used for the 2-by-2 Cube 118

ACKNOWLEDGMENTS

First and foremost, I would like to thank my advisor, Risi Kondor, for his kindness, support and mentorship throughout my time at UChicago. I'm reminded of one of Geoff Hinton's famous answers from his Reddit AMA years ago: "Some people [...] can actually crank a mathematical handle to arrive at new insights. I cannot do that." Risi certainly can and it has been a wonderful privilege to have had a front row seat to his magic during my time at UChicago. Though I have seen the trick many times at this point, it still amazes me whenever Risi's ideas work out exactly as he said it would. One of the things I will miss the most about the PhD experience is our conversations (research and otherwise) during our one-on-ones.

Thank you to Eric Jonas and Yuxin Chen, my other committee members for their patience in understanding my research and their helpful comments/feedback. I am grateful to Eric who served as a surrogate academic parent and provided a much needed semblance of normality while Risi was on sabbatical at the Flatiron during the fall of 2019. Our conversations always left me feeling optimistic and energized. I remember being a part of the grad student lunch with Yuxin when he was interviewing at UChicago. He was just as kind and gracious with his time then as he is now.

I would also like to express my gratitude and appreciation to my fellow labmates, and department mates over the years: Yi Ding, Pramod Mudrakarta, Jonathan Eskreis-Winkler, Hy Truong Son, Byol Kim, Richard Zhu, Nancy Cheng, Shubhendu Trivedi, Yulie Zamora, Liwen Zhang, and Erik Thiede. Special thanks to Brandon Anderson, one of our postdocs, who brought a wonderful warmth to our group during his time with us.

Finally, I'd like to thank my parents and my sister for standing by me this entire time and Christine for her love and support. Grad school has been a long journey, perhaps a bit too long but I am grateful for all the friends I've made and experiences I've had along the way.

ABSTRACT

In this thesis we describe two separate works: higher order permutation equivariant layers for neural networks and Fourier bases for reinforcement learning over combinatorial puzzles.

Recent work on permutation equivariant neural networks has mostly focused on the first order case (sets) and the second order case (graphs). We describe the machinery for generalizing permutation equivariance to arbitrary k-ary interactions and provide a systematic approach for efficiently computing these k'th-order permutation equivariant layer and enumerating all the intermediate operations involved. We evaluate our proposed permutation equivariant architectures using higher order equivariant layers on a variety of set learning tasks and find that our models are competitive with existing baseline methods while using much fewer parameters.

Traditionally, permutation puzzles such as the Rubik's Cube were often solved by heuristic search like A*-search and value based reinforcement learning methods. Both heuristic search and Q-learning approaches to solving these puzzles can be reduced to learning a heuristic/value function to decide what puzzle move to make at each step. We propose learning a value function using the irreducible representations basis (which we will also call the Fourier basis) of the puzzle's underlying group. Classical Fourier analysis on real valued functions tells us we can approximate smooth functions with low frequency basis functions. Similarly, smooth functions on finite groups can be represented by the analogous low frequency Fourier basis functions. We demonstrate that we can learn effective value functions in the Fourier basis for solving various permutation puzzles using fewer parameters and fewer samples than deep value networks.

SUMMARY

The work presented in this thesis has appeared in the following peer-reviewed publications:

1. Pan, Horace, and Risi Kondor. "Permutation Equivariant Layers for Higher Order Interactions." International Conference on Artificial Intelligence and Statistics, pp. 5987–6001. PMLR, 2022.
2. Pan, Horace, and Risi Kondor. "Fourier Bases for Solving Permutation Puzzles." In International Conference on Artificial Intelligence and Statistics, pp. 172-180. PMLR, 2021.

Apart from the material presented in this thesis, I was also a core contributor to the following works during my graduate studies:

1. Kondor, Risi, and Horace Pan. "The Multiscale Laplacian Graph Kernel." Advances in Neural Information Processing Systems 29 (2016).
2. Kondor, Risi, Hy Truong Son, Horace Pan, Brandon Anderson, and Shubhendu Trivedi. "Covariant compositional networks for learning graphs." arXiv preprint arXiv:1801.02144 (2018).
3. Hy, Truong Son, Shubhendu Trivedi, Horace Pan, Brandon M. Anderson, and Risi Kondor "Predicting Molecular Properties with Covariant Compositional Networks." The Journal of Chemical Physics 148.24 (2018): 241745.

Part I

Permutation Equivariant Layers for Higher Order Interactions

CHAPTER 1

INTRODUCTION

In machine learning, we often deal with data that exhibit some form of **symmetry**. Loosely, our data has a certain symmetry when we can apply some transformation to our data without changing some property of that data. A common example of this is rotational and translational symmetry in images. Applying rotations and translations to an image (generally) does not affect the contents of that image. Data that comes with 3-dimensional coordinates

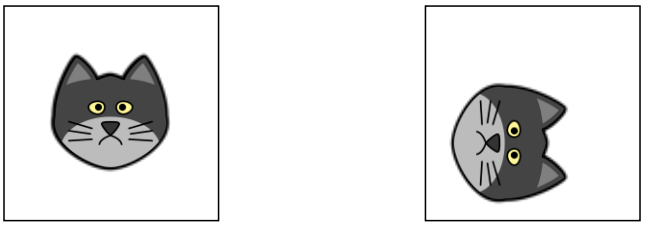


Figure 1.1: Images exhibit rotational and translational symmetry.

such as point cloud data (Figure 1.2b) may exhibit 3D rotational symmetry; the points of the point cloud can be rotated as a group while maintaining the integrity of the overall shape. If we have a model that takes as inputs data that exhibit these symmetries, we would like the output of the model to remain the same after some of these transformations, as the underlying object has not changed after these rotations or translations. The central question then, is how do we construct machine learning models that respect these transformations of our data?

We are interested in **permutation symmetry**, which often arises when we work with discrete objects such as graphs and sets. If we are interested in constructing machine learning models that take as input permutation symmetric data, we would like models to treat the data the same no matter how the inputs are ordered. Similarly, items in a set have no inherent order and can be presented in any order without affecting properties of the collection of items. Consider the following motivating examples for permutational symmetry in data:

(a) (b)

Figure 1.2: Molecules (1.2a) and point clouds (1.2b) exhibit 3D rotational symmetry. Vertices of the graph and points of the point cloud have permutation symmetry.

- Suppose we have a collection of images taken of a single animal. From this collection of images, we would like to infer certain characteristics of the animal such as its age, species, breed. The ordering of the images in the collection should not affect our predictions. If there is only one image then the task is a standard image classification/regression. We have multiple photos of the same animal and we would like to somehow use all of the information present to produce our predictions.

- In graph classification or regression we want to map a given graph $G = (V, E)$ to a target value. The ordering of the vertices in the vertex set $V = \{v_1, v_2, \ldots, v_n\}$ can be arbitrarily permuted without changing the structure of the graph or its corresponding target value.

- In point cloud data, we might have sets of points in three dimensions along with features associated with each point. We might be interested in classifying the shape of the point cloud. As with graphs, the ordering of these points does not affect the associated target class of the point cloud.

As we will come to see, central to constructing neural networks that respect permutation symmetries is the concept of **permutation invariance and equivariance**. We will explore

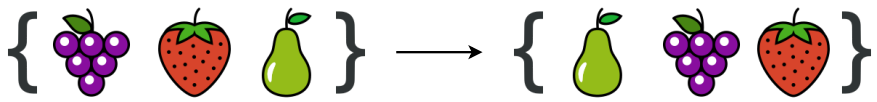


Figure 1.3: Items in a set have no inherent ordering and can be permuted arbitrarily without affecting their composition.

different ways of constructing such permutation equivariant layers.

1.1 Preliminaries/Notation

We begin by formally defining some mathematical terms and expressions that will appear throughout subsequent sections.

1.1.1 Groups

It turns out that **groups** are exactly the right mathematical abstraction for talking about symmetries of objects.

Definition 1.1.1 (Group)**.** *A group is a set of elements $\mathcal{G}$ that comes endowed with a group operation $\cdot : \mathcal{G} \times \mathcal{G} \to \mathcal{G}$ that satisfies the following three properties:*

- *There exists an identity element of the group denoted e, such that for any $g \in \mathcal{G}$, $e \cdot g = g \cdot e = g$.*
- *For any $g \in \mathcal{G}$, there exists an inverse of g denoted g^{-1} such that $g \cdot g^{-1} = g^{-1} \cdot g = e$.*
- *The group operation is associative: for any $a, b, c \in \mathcal{G}$, $a \cdot (b \cdot c) = (a \cdot b) \cdot c$.*

Definition 1.1.2 (Cyclic group)**.** *The cyclic group of order n denoted C_n is the set of integers $\{0, 1, \ldots, n-1\}$, with the group operation being addition modulo n. For $a, b \in C_n$, $a \cdot b = (a + b) \mod n$.*

Definition 1.1.3 (Symmetric group)**.** *The symmetric group of degree n, denoted $\mathbb{S}_n$, is the set of all bijections from $\{1, 2, \ldots, n\}$ to $\{1, 2, \ldots, n\}$. The group operation is function composition.*

Definition 1.1.4 (Permutation)**.** *A permutation $\pi \in \mathbb{S}_n$ is a bijective function from $\{1, 2, \ldots, n\}$ to $\{1, 2, \ldots, n\}$.*

It is straightforward to see that the symmetric group does indeed follow the three required conditions to be a group. The identity element is the permutation that sends $i \mapsto i$ for $i = 1, 2, \ldots, n$. Every permutation has an inverse which is simply the permutation that undoes the original mapping. Finally, function composition is associative.

Since we will be using permutations over and over (in Part II as well), it will be convenient to be able to write them in a standard concise notation. The notation that is often used to write permutations is the cycle decomposition format.

Definition 1.1.5 (Cycle Decomposition)**.** *The cycle decomposition of a permutation is a collection of cyclic tuples where consecutive entries of the cyclic tuple encode the mapping of the permutation. The cycle $(a_1, a_2, \ldots, a_k)$ indicates the part of the permutation that sends $a_1 \mapsto a_2, a_2 \mapsto a_3, \ldots, a_k \mapsto a_1$.*

If a permutation π maps a number i to itself, we usually omit the singleton cycle (i) in the cycle decomposition of π. When discussing permutations, we will generally explicitly state the order of the permutation group it is coming from if it is not clear from context.

Example 1. *The permutation $((1, 2), (3, 4)) \in \mathbb{S}_4$ is the permutation π such that:*

$$\pi(1) = 2, \quad \pi(2) = 1, \quad \pi(3) = 4, \quad \pi(4) = 3.$$

Example 2. *The permutation $((1, 3), (2, 5)) \in \mathbb{S}_5$ is the permutation π such that:*

$$\pi(1) = 3, \quad \pi(2) = 5, \quad \pi(3) = 1, \quad \pi(4) = 4, \quad \pi(5) = 2.$$

Example 3. $\mathbb{S}_3$*, the Symmetric group of order* 3*, is comprised of the following six permutations:*

$$\mathbb{S}_3 = \{e, (1,2), (1,3), (2,3), (1,2,3), (1,3,2)\}.$$

Up to this point we have only discussed permutations and the symmetric group, which are not the objects that are being passed to our machine learning models. We are instead interested in a collection of items or objects that can be permuted or relabeled amongst themselves.

Definition 1.1.6 (Group actions)**.** *Given a group $\mathcal{G}$, and a set $\mathcal{E}$, the (left)* ***group action*** *of $\mathcal{G}$ on $\mathcal{E}$ is a function $\alpha : G \times \mathcal{E} \to \mathcal{E}$ that satisfies the two conditions:*

- *Identity: $e \cdot x = x$ for all $x \in \mathcal{E}$*
- *Compatibility: $\alpha(g, \alpha(h, x)) = \alpha(gh, x)$ for all $g, h \in \mathcal{G}$ and $x \in \mathcal{E}$*

For notational convenience we will often write $g \cdot x$ or gx instead of $\alpha(g, x)$ to when discussing group actions.

We will primarily be discussing permutation actions on various objects such as sets and what we refer to as **permutable tensors**.

Suppose we have a collection of entities that have interactions amongst themselves. The collection can be described using a graph. In graphs, the set of entities is the vertex set. Edges of the graph encode pairwise interactions. Beyond pairwise interactions, we may also have k-ary interactions captured by k-hyperedges, which indicate some relationship between k of the vertices. It is natural to consider higher order interactions between entities in other contexts such as sets, which may not explicitly be endowed with these k-ary hyperedges. To formalize the idea of tensors that encapsulate the information about k-ary interactions between elements of our base set $\mathcal{E}$, we define **permutable k'th order tensors**.

Definition 1.1.7 (Permutable k'th order tensors)**.** *Given a set of entities $\{e_1, \ldots, e_n\} \in \mathcal{E}$, a k'th order permutable tensor $X \in \mathbb{R}^{n^k \times d}$ is a multidimensional array. The last dimension of X we call the feature dimension.*

We specifically call these "permutable" tensors because if the entities are relabeled by some permutation $\pi \in \mathbb{S}_n$, then the corresponding entries of the permutable tensor are permuted. Let $\pi \cdot X$ denote the resulting tensor after the entities are mapped from $\{e_1, \ldots, e_n\} \mapsto \{e_{\pi(1)}, \ldots, e_{\pi(n)}\}$. We get: $X \mapsto \pi \cdot X$ where the entries of $\pi \cdot X$ are:

$$[\pi \cdot X]_{i_1,\ldots,i_k} = [X]_{\pi^{-1}(i_1),\ldots,\pi^{-1}(i_k)}.$$

$X \in \mathbb{R}^{n^k \times d}$ can be viewed as a multidimensional array that contains information about the k-ary relationships between the entities. X can be indexed by k-tuples in $\{1, 2, \ldots, n\}^k$. If we are interested in the interaction between the k entities $(e_{i_1}, e_{i_2}, \ldots, e_{i_k})$ (not necessarily unique), then $X_{i_1,\ldots,i_k} \in \mathbb{R}^d$ can be interpreted as a feature vector for the k-ary interaction between $e_{i_1}, e_{i_2}, \ldots, e_{i_k}$.

Example 4. *Let $x_1, x_2, \ldots, x_n \in \mathbb{R}^d$ be d-dimensional feature vectors corresponding to objects $e_1, \ldots, e_n$. After stacking the x_i vectors into a matrix $X \in \mathbb{R}^{n \times d}$, we get a first order permutable tensor with d channels (or d features):*

$$X = \begin{bmatrix} x_1^\top \\ x_2^\top \\ \vdots \\ x_n^\top \end{bmatrix}.$$

Example 5. *Let G be a graph: $G = (V, E)$ with adjacency matrix $A \in \mathbb{R}^{n \times n}$. The adjacency matrix can be viewed as a 2nd order permutable tensor, where the feature dimension size is one. The Symmetric group of order n, $\mathbb{S}_n$, acts on the adjacency matrix of G by permuting*

its rows and columns: $[\pi \cdot A]_{i,j} = [A]_{\pi^{-1}(i),\pi^{-1}(j)}$ *for all* $1 \leq i, j \leq n$.

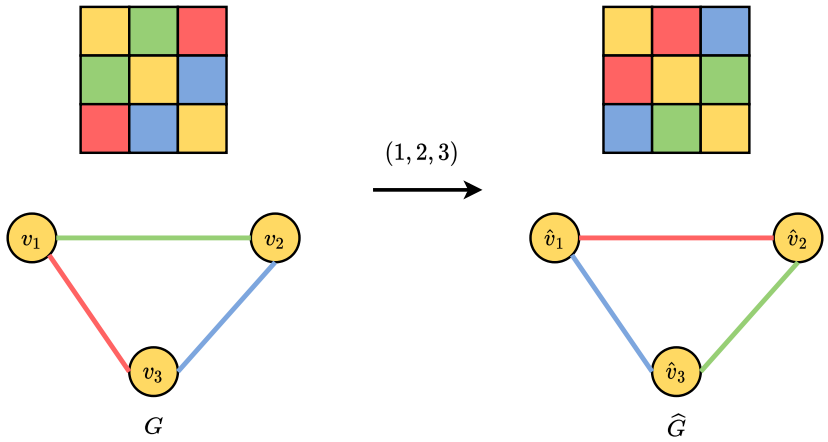


Figure 1.4: Relabeling vertices v_1, v_2, v_3 with permutation $\pi = (1, 2, 3) \in \mathbb{S}_3$ induces the change in the graph and adjacency matrix (a 2nd order permutable tensor) above.

1.1.2 Neural Networks

The type of machine learning models we will be considering for modeling data that exhibits permutational symmetries are neural networks. A neural network is a nonlinear function $f : \mathcal{X} \to \mathcal{Y}$, where the input space $\mathcal{X}$ is usually a vector space (such as $\mathbb{R}^d$ for some dimension d) and $\mathcal{Y} = \mathbb{R}$ for regression or $\mathcal{Y} = \{1, 2, \ldots, K\}$ for multiclass classification. Modern neural networks can succinctly be described as a sequence of interleaved linear(or affine functions) and nonlinear functions:

$$f(x) = \sigma \circ L_N(\sigma \circ L_{N-1}(\ldots(\sigma \circ L_1(x)))).$$

In the above expression, the L_i's denote linear functions and σ denotes the chosen nonlinear function. Typically the nonlinear function is an elementwise nonlinearity such as the rectified linear unit or hyperbolic tangent function. The choice of nonlinearity does not affect performance much so most practitioners default to using ReLUs as it is computationally

cheaper than the alternatives. In Part I, we will be interested in neural networks that take permutable tensors as input.

1.1.3 Permutation Equivariance

We have formally defined permutable tensors and the ways that permutations interact with these structured tensors. Now we introduce permutation invariance and equivariance. Equivariance is now understood to be a useful guiding principle for designing neural networks when our data exhibits certain symmetries [Miller et al., 2020]. This benefit to equivariant networks is well known in the case of computer vision with convolutional neural networks, whose intermediate layers are translation equivariant. Equivariant layers have since been shown to provide better parameter efficiency and sample efficiency in a wide variety of domains where we require equivariance with respect to a symmetry group [Cohen and Welling, 2016, Linmans et al., 2018, Worrall and Brostow, 2018, Batzner et al., 2022].

In general, for a symmetry group $\mathcal{G}$ an input space $\mathcal{E}_1$ and output space $\mathcal{E}_2$ such that $\mathcal{E}_1, \mathcal{E}_2$ both have associated $\mathcal{G}$-actions, a $\mathcal{G}$-invariant function $f : \mathcal{E}_1 \to \mathcal{E}_2$ must satisfy:

$$f(g \cdot X) = f(X)$$

for all $g \in \mathcal{G}, X \in \mathcal{E}_1$. f is equivariant with respect to $\mathcal{G}$-actions if the following commutative diagram holds:

$$\begin{array}{ccc} \mathcal{E}_1 & \xrightarrow{g} & \mathcal{E}_2 \\ {\scriptstyle f}\downarrow & & \downarrow{\scriptstyle f} \\ \mathcal{E}_1 & \xrightarrow{g} & \mathcal{E}_2 \end{array}$$

In other words:

$$g \cdot f(X) = f(g \cdot X).$$

When the symmetry group $\mathcal{G}$ is the symmetric group $\mathbb{S}_n$, we have permutation invariance

and equivariance. For concreteness, we generally have our input space $\mathcal{E}_1$ being k_1'th order permutable tensors and the output space $\mathcal{E}_2$ being k_2'th order permutable tensors.

Definition 1.1.8 (Permutation Invariance)**.** *Let f be a function that takes as input a k_1'th order permutable tensor $f : \mathbb{R}^{n^{k_1} \times d_1} \to \mathbb{R}^{n^{k_2} \times d_2}$. f is said to be permutation invariant if*

$$f(\pi \cdot X) = f(X)$$

for all $X \in \mathbb{R}^{n^{k_1} \times d}$, $\pi \in \mathbb{S}_n$.

Definition 1.1.9 (Permutation Equivariance)**.** *Let f be a function from k_1'th order permutable tensors to k_2'th order permutable tensors: $f : \mathbb{R}^{n^{k_1} \times d_1} \to \mathbb{R}^{n^{k_2} \times d_2}$ is said to be* ***permutation equivariant*** *if*

$$f(\pi \cdot X) = \pi \cdot f(X),$$

for all permutations $\pi \in \mathbb{S}_n, X \in \mathbb{R}^{n^{k_1} \times d}$.

Note that permutation invariance is just a special case of permutation equivariance, where the permutation acts trivially on the output tensor. We will generally want to use a permutation invariant operation to bring a k'th order tensor down to a 0'th order tensor, which will effectively be a feature vector/representation vector for the set of items. Then this 0'th order tensor can be used as input in multilayer perceptron or linear layer to produce an output prediction.

Example 6. *Given a graph $G = (V, E)$ with adjacency matrix $A \in \mathbb{R}^{n \times n}$, let $f : \mathbb{R}^{n \times n} \to \mathbb{R}^n$ be the function that takes as input an adjaceny matrix and returns a vector of node degrees of the graph:*

$$[f(A)]_i = \text{degree}(v_i) = \sum_j A_{ij}.$$

Then f is a permutation equivariant function that takes a $2nd$ order permutable tensor as input and returns a $1st$ order permutable tensor.

Example 7. *Given a set of objects* $\{e_1, \ldots, e_n\}$*, let the third order permutable tensor* $X \in \mathbb{R}^{n^3 \times d}$ *describe* 3*-ary interactions between the given objects. So* $X_{ijk} \in \mathbb{R}^d$ *is a* d*-dimensional feature vector associated with the triplet* $\{e_i, e_j, e_k\}$*. The function* $f : \mathbb{R}^{n^3 \times d} \to \mathbb{R}^{n^2 \times d}$ *given by:* $[f(X)]_{ij} = \sum_k X_{ijk}$ *is a function mapping* 3*rd order permutable tensors to* 2*nd order permutable tensors.*

Example 8. *Let* $X \in \mathbb{R}^{n^3 \times d}$*. Let* $f : \mathbb{R}^{n^3 \times d} \to \mathbb{R}^d$ *be the function that sums an input* 3*rd order permutable tensor over the three entity indices:* $[f(X)]_l = \sum_{i,j,k} X_{ijkl}$*, for* $l = 1, 2, \ldots, d$*. Then* f *is a permutation invariant function.*

Operations that act "row-wise" on a permutable tensor are permutation equivariant. A row-wise operation on a k'th order permutable tensor is one that processes each k-entity slice independently and identically. For instance, let $X \in \mathbb{R}^{n^k \times d_1}$ be our input k'th order permutable tensor, $h : \mathbb{R}^{d_1} \to \mathbb{R}^{d_2}$ be a multilayer perceptron. A neural network layer $H : \mathbb{R}^{n^k \times d_1} \to \mathbb{R}^{n^k \times d_2}$ that simply applies h to each k-slice of X is a permutation equivariant function:

$$[H(X)]_{i_1,\ldots,i_k} = h(X_{i_1,\ldots,i_k}).$$

It is straightforward to see that applying row-wise functions to permutable tensors is a permutation equivariant operation. For any $\pi \in \mathbb{S}_n$, we have:

$$\begin{aligned}
[\pi \cdot H(X)]_{i_1,\ldots,i_k} &= [H(X)]_{\pi^{-1}(i_1),\ldots,\pi^{-1}(i_k)} \\
&= h(X_{\pi^{-1}(i_1),\ldots,\pi^{-1}(i_k)}) \\
&= [H(\pi \cdot X)]_{i_1,\ldots,i_k} \\
\pi \cdot H(X) &= H(\pi \cdot X)
\end{aligned}$$

As a special case of this, elementwise nonlinearities are also permutation equivariant operations. So if row-wise operations are already permutation equivariant, why bother trying to construct other kinds of permutation equivariant functions? In the row-wise multilayer

perceptron example above, each k-slice $X_{i_1,\ldots,i_k}$ is only updated by information about itself; it does not take into account any of the k-ary interactions between any of the other entities. A k'th order permutable tensor contains a lot of information that is relevant for any k-subset of elements that is not being used if we just use row-wise operations.

1.2 Problem Statement

We are interested in constructing permutation equivariant linear functions that take as input k_1'th order permutable tensors of some feature dimension size d_1 and return k_2'th order permutable tensors with some arbitrary output feature dimension size d_2. Formally, we would like to construct linear functions L where

$$L : \mathbb{R}^{n^{k_1} \times d_1} \to \mathbb{R}^{n^{k_2} \times d_2}.$$

The main questions we are interested in and will try to answer are:

- what form can permutation equivariant linear functions L take?
- what are some guiding principles and considerations for constructing permutation equivariant networks?
- how effective are these permutation equivariant architectures on set learning tasks?

CHAPTER 2

RELATED WORK

Before we discuss our constributions, we first take a tour through (what we view as) the foundational ideas in the machine learning literature related to permutation invariance and equivariance in neural networks.

2.1 Early Permutation Invariant Architectures

Among the earliest to articulate the conditions that permutation invariant architectures should obey was PointNet [Qi et al., 2017] and Deep Sets[Zaheer et al., 2017]. This is not to say that the architectures proposed by them were entirely novel. Various concurrent works also described permutation invariant neural network architectures for applications including multi-agent reinforcement learning [Sukhbaatar et al., 2016], dynamics predictions of colliding bodies [Guttenberg et al., 2016], and learning population level statistics [Edwards and Storkey, 2016], etc. PointNet [Qi et al., 2017] and Deep Sets [Zaheer et al., 2017] are particularly notable for their seminal contributions to permutation invariant and equivariant neural network designs.

2.1.1 PointNet

PointNet, introduced by Qi et al. [2017], is a neural network architecture for segmentation or multiclass classification on point clouds. A point cloud is a collection of 3D points: $\{p_i \in \mathbb{R}^3 \mid i = 1, \ldots, n\}$ used to represent a 3D object. The points are often generated by sampling the object in question. Practictioners may be interested in classifying the point cloud or segmenting the point cloud. The main properties of point clouds is that the points of the point cloud are unordered and should be treated as a set. Point clouds should also be rotation and translation invariant: applying the same rotation or translation to every point

in the point cloud should not affect any representation the point cloud.

PointNet learns a featurization of the point cloud by applying a nonlinear neural network on each point individually first before aggregating the collection points' representations with a max pooling operation. The full architecture of PointNet is shown in Fig 2.1. PointNet's architecture can be summarized as the following function: $f : \mathbb{R}^{n\times 3} \to \{1, 2, \ldots, C\}$, where C is the number of classes.

$$f(x_1, x_2, \ldots, x_n) = \psi(\text{MAX}\{\phi(x_1), \phi(x_2), \ldots, \phi(x_n)\}).$$

where $\phi : \mathbb{R}^3 \to \mathbb{R}^{1024}$ is the shared neural network used to encode each individual point, and $\psi : \mathbb{R}^{1024} \to \mathbb{R}^{|C|}$ is the neural network that takes in the 1024 dimensional point cloud encoding to produce the final classification.

PointNet also presents a theorem which effectively states that their proposed architecture is expressive enough to approximate any set function.

Theorem 1 (Qi et al. [2017])**.** *Suppose $f : \mathcal{X} \to \mathbb{R}$ is a continuous set function with respect to Hausdorff distance $d_H(\cdot, \cdot)$. For all $\epsilon > 0$, there exists a continuous function h and a symmetric g such that $g(x_1, \ldots, x_n) = \gamma \circ \text{MAX}(\{x_1, \ldots, x_n\})$ such that for any $S \subseteq \mathcal{X}$*

$$|f(S) - \gamma(\text{MAX}\{h(x_i) \mid x_i \in S\})| < \epsilon$$

where $x_1, \ldots, x_n$ is the full list of elements in S, γ is a continuous function, and MAX *is a vector max operator that takes n vectors as input and returns a new vector of the element-wise maximum.*

Aside from showing compelling results on a variety of point cloud learning tasks and 3D semantic segmentation tasks, PointNet remains influential because it was the first work to provide the following prescription for designing permutation invariant networks:

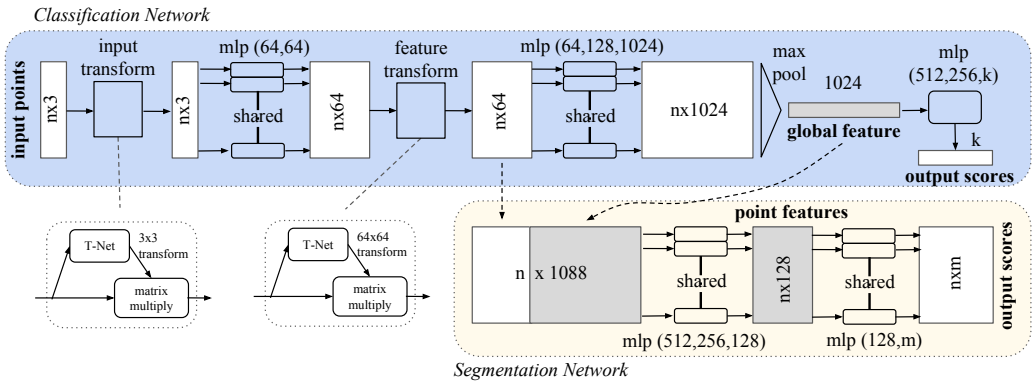


Figure 2.1: Figure from Qi et al. [2017]. Each point of the point cloud is input as a 3 dimensional vector that is subsequential encoded by nonlinear transformations. The collection of 1024-dimensional representations of the points is aggregated with a max pooling operation to generate a global feature vector for the point cloud. Finally, this global feature vector goes through a final multilayer perceptron to produce the output classification of the network.

- each element of a set can be encoded individually with a shared neural network

$$\phi : x_i \mapsto \phi(x_i)$$

- the encoded elements are aggregated with a symmetric function (e.g.: channel-wise max, mean, or sum) to yield a global feature representation:
- the global feature representation is passed through a final neural network to produce a prediction

2.1.2 Deep Sets

Deep Sets [Zaheer et al., 2017] was the first to formalize and articulate a necessary and sufficient permutation equivariant architecture for first order permutable tensors. Similarly to the learning task in PointNet, Zaheer et al. [2017] considers set classification problems.

In this regime, our dataset D looks like the following:

$$D = \{(S_1, y_1), (S_2, y_2), \ldots, (S_{|D|}, y_{|D|})\}$$

and $S_i = \{x_1, x_2, \ldots, x_{l_i}\}$, where $l_i = |S_i|$. Furthermore, each $x_i \in \mathbb{R}^d$ and $y \in \mathbb{R}$ for regression tasks and $y \in \{1, 2, \ldots, k\}$ for classification tasks.

A crucial property of this collection of items is that the order in which they appear does not matter. If we reordered the objects in a set with a permutation $\pi \in \mathbb{S}_n$, so that $\{e_1, \ldots, e_n\} \mapsto \{e_{\pi(i)}, \ldots, e_{\pi(n)}\}$, the associated target value should not change. This **permutation symmetry** in our data should also be manifested in our machine learning models somehow. Zaheer et al. [2017] first characterizes the form of permutation invariant functions with the following theorem:

Theorem 1. *A function f operating the set $\{x_1, \ldots, x_n\}$ with elements from a countable collection, is invariant to the permutations of items in the set, if and only if it can be decomposed in the form $f(\{x_1, \ldots, x_n\}) = \rho(\sum_i \phi(x_i))$ for suitable functions ρ and ϕ.*

Generally our data for sets are input as first order permutable tensors $X \in \mathbb{R}^{n \times d}$ where the ith row of X corresponds to the d-dimensional feature vector describing the ith item. Zaheer et al. [2017] also characterizes the form of any permutation equivariant linear layer on first order permutable tensors.

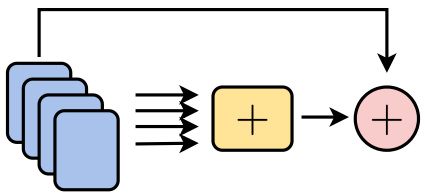


Figure 2.2: On input $x_1, x_2, \ldots, x_n \in \mathbb{R}^d$, a Deep Sets permutation equivariant layer outputs a linear combination of the individual x_i's and their sum $\sum_j x_j$.

Theorem 2. *Given input $X \in \mathbb{R}^{m \times d}$, a linear function $f_\Theta : \mathbb{R}^m \to \mathbb{R}^m$ which can be written as $f_\Theta(X) = \Theta X$, $\Theta \in \mathbb{R}^{m \times m}$ is* ***permutation equivariant*** *if and only if the off diagonal elements of Θ are tied together and the diagonal elements are equal as well. In other words, Θ must have the following form:*

$$\Theta = \lambda \mathbf{I}_m + \gamma(\mathbf{1}_m \mathbf{1}_m^\top) \tag{2.1}$$

where $\lambda, \gamma \in \mathbb{R}$ and $\mathbf{I}_m$ is the $m \times m$ identity matrix and $\mathbf{1}_m \in \mathbb{R}^{m \times 1}$ is a vector of ones.

This means for any x_i, the result of a linear permutation equivariant layer must be:

$$x_i \mapsto \lambda x_i + \gamma \sum_j x_j.$$

A composition of permutation equivariant functions is still permutation equivariant and elementwise nonlinearities will also preserve permutation equivariance. Thus, a permutation invariant neural network can be constructed by interleaving a series of permutation equivariant layers and elementwise nonlinearities before applying a final permutation invariant layer. Zaheer et al. [2017] proves that the only kind of permutation equivariant linear layer from first order permutable tensors to first order permutable tensors must be of the form given in Eqn (2.1).

For completeness, we show the proof of equivariance of Eqn 2.1 to demonstrate the general idea behind equivariance proofs. We must show that the matrix Θ commutes with any permutation matrix. This can usually be proved directly.

Theorem 2. *$f_\Theta(X) = (\lambda \mathbf{I}_m + \gamma(\mathbf{1}_m \mathbf{1}_m^\top))(X)$ is a permutation equivariant function.*

Proof. Consider an arbitrary index i. $[\pi \cdot X]_i = X_{\pi^{-1}(i)}$. On the other hand

The ith index of $f_\Theta(\pi \cdot X)$ is:

$$\begin{aligned}
[f_\Theta(\pi \cdot X)]_i &= \lambda[\pi \cdot X]_i + \gamma[\mathbf{1}_m \mathbf{1}_m^\top (\pi \cdot X)]_i \\
&= \lambda[X]_{\pi^{-1}(i)} + \gamma \sum_j X_{\pi^{-1}(j)} \\
&= \lambda[X]_{\pi^{-1}(i)} + \gamma \sum_j X_j
\end{aligned}$$

This is the same as the ith index of $\pi \cdot f_\theta(X)$:

$$\begin{aligned}
[\pi \cdot f_\theta(X)]_i &= \lambda X_{\pi^{-1}(i)} + \pi \cdot (\gamma \mathbf{1}_m \mathbf{1}_m^\top X) \\
&= \lambda[X]_{\pi^{-1}(i)} + \gamma \sum_j X_j
\end{aligned}$$

Alternatively, we can also see that for any $\pi \in \mathbb{S}_n$ and its associated permutation matrix P_π, the permutation matrix commutes with the identity and a permutation matrix commutes with the matrix of all ones:

$$\begin{aligned}
\boldsymbol{P}_\pi \mathbf{I}_m &= \mathbf{I}_m \boldsymbol{P}_\pi \\
\mathbf{1}_m \mathbf{1}_m^\top \boldsymbol{P}_\pi &= \boldsymbol{P}_\pi \mathbf{1}_m \mathbf{1}_m^\top
\end{aligned}$$

Thus, f_Θ must be permutation equivariant. □

Notice that any permutation invariant function can be used in place of the $\mathbf{1}_m \mathbf{1}_m^\top$ operator in Equation (2.1) and we still have a permutation equivariant layer. For instance: $f(X) = \lambda X + \gamma \text{maxpool}(X)$, where maxpool : $\mathbb{R}^{n \times d} \to \mathbb{R}^d$ computes a row-wise maximum. The permutation invariant operation in the layer is often referred to as a pooling operation (ex: sum pooling, max pooling, mean pooling, etc). Sum pooling is most commonly used in Deep Sets styled architectures.

Zaheer et al. [2017] also conveys that any applying successive permutation equivariant

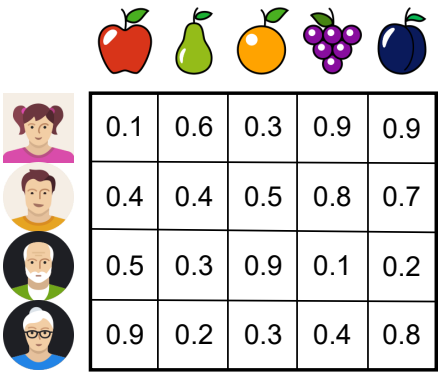
layers will still retain permutation equivariance. We saw earlier that elementwise nonlinearities were also permutation equivariant. Thus, we can construct arbitrarily large permutation equivariant neural networks by simply interleaving permutation equivariant layers, elementwise nonlinearities before taking a permutation invariant summation.

2.2 Parameter Sharing and Permutation Equivariance

Deep Sets only consider equivariance with respect to what we call first order permutation actions. However, one might imagine having data that transform with permutations in more complicated ways. As a motivating example, consider user-item affinity matrix (ex: Fig 2.3) $X \in \mathbb{R}^{N \times M}$, where we have N users and M items. X_{ij} denotes the score that user i assigns to item j. In this setting, we might consider having feature tensors $X \in \mathbb{R}^{N \times M \times d_{in}}$ and linear permutation equivariant layers. That is, functions $\phi : \mathbb{R}^{N \times M \times d_{in}} \to \mathbb{R}^{N \times M \times d_{out}}$. Users can be relabeled/reordered in any arbitrary way, and likewise, movies can be reordered in any arbitrary way giving us permutation symmetry across users and across movies. Similar to the previous sections, we are interested in constructing permutation equivariant layers for these user-movie interaction matrices. Recall that Deep Sets permutation equivariant layers had linear weights that effectively only had two parameters. It turns out that the permutation equivariant layers for relational data, where the data has permutational symmetry across multiple axes, also has drastic parameter sharing restrictions on the equivariant linear layers.

Before we describe the key findings and theorems of Hartford et al. [2018], we first introduce some of their notation, which is also used in Graham et al. [2019]. Let vec denote the operator that takes in a matrix and returns the matrix in its vectorized form and vec^{-1} denote the inverse of the vectorization that takes in a vector and returns it as a matrix so that $\text{vec}^{-1}(\text{vec}(X)) = \text{vec}(\text{vec}^{-1}(X)) = X$. Hartford et al. [2018] introduce an "exchangeable" matrix layer for situations where the rows and columns of a matrix are interchangeable.

Definition 2.2.1. *Given $X \in \mathbb{R}^{N \times M}$, a fully connected layer $\sigma(W\text{vec}(X))$ with $W \in$*



0.1	0.6	0.3	0.9	0.9
0.4	0.4	0.5	0.8	0.7
0.5	0.3	0.9	0.1	0.2
0.9	0.2	0.3	0.4	0.8

Figure 2.3: User-Item affinity data. Each entry of the matrix is a score for how much a user enjoys the given fruit.

$\mathbb{R}^{NM\times NM}$ *is called an exchangeable matrix layer if, for all permutations* $\pi \in \mathbb{S}_N$ *permutations* $\tau \in \mathbb{S}_M$ *and an elementwise nonlinearity* σ *(e.g.: ReLU), permutation of the rows by* π *and columns by* τ *results in the same permutations applied to the output of the layer:*

$$\text{vec}^{-1}(\sigma(W\text{vec}(P_\pi X P_\tau))) = P_\pi \text{vec}^{-1}(\sigma(W\text{vec}(X)))P_\tau$$

Let $Y \in \mathbb{R}^{N\times M}$ be the output of of the "exchangeable matrix layer". Abusing notation slightly, we denote entries of W using pairs of tuples. The first element of the tuple indicates the relevant output index in Y and the latter tuple element indicates the input index from X:

$$Y_{n,m} = \sum_{n'\in[N],m'\in[M]} W_{(n,m),(n',m')} X_{n',m'}.$$

Hartford et al. [2018] shows that the permutation equivariance condition described in 2.2.1's

results give us the following parameter sharing scheme for W:

$$W_{(n,m),(n',m')} \coloneqq \begin{cases} w_1 & , \text{ if } n = n', m = m' \\ w_2 & , \text{ if } n = n', m \neq m' \\ w_3 & , \text{ if } n \neq m', m = m' \\ w_4 & , \text{ if } n \neq n', m \neq m' \end{cases}$$

The implication of this parameter sharing scheme is that a permutation equivariant layer for user-item data tensors can be written more compactly as the following linear combination of symmetric sums of rows and column entries:

$$Y = \sigma(w_1 X + \frac{w_2}{N}(\mathbf{1}_N \mathbf{1}_N^\top X) + \frac{w_3}{M}(X \mathbf{1}_M \mathbf{1}_M^\top) + \frac{w_4}{NM}(\mathbf{1}_N \mathbf{1}_N^\top X \mathbf{1}_M \mathbf{1}_M^\top) + w_5(\mathbf{1}_N \mathbf{1}_M^\top))$$

The expression above is a bit more digestable when focusing on a specific i, j index:

$$Y_{ij} = w_1 X_{ij} + w_2 \sum_l X_{lj} + w_3 \sum_k X_{ik} + w_4 \sum_{kl} X_{kl} + w_5.$$

The output feature vector for the pair (i, j) is a linear combination of: the input (i, j) feature vector, features summed over the columns, features summed over the rows, features summed globally, and a constant bias term.

If we instead were interested in the relationship between more than two items, we might have a relational table over more than two sets of entities. Hartford et al. [2018] also consider the parameter sharing scheme for data tensors of arbitrary arity. Suppose, the data tensor were of the form $X \in \mathbb{R}^{N_1 \times N_2 \times \ldots N_k \times d}$, where we have sets of objects $\mathcal{X}_1, \ldots, \mathcal{X}_k$ with $|\mathcal{X}_i| = N_i$ for all i. Any model that takes as input a multidimensional array of shape $\mathbb{R}^{N_1 \times \ldots \times N_k \times d_1}$ and returns tensor of shape $\mathbb{R}^{N_1 \times \ldots \times N_k \times d_2}$ should be permutation equivariant with respect to permutations along any of the individual entity dimensions (the first k dimensions). Graham

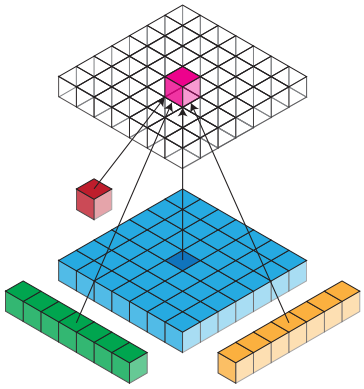


Figure 2.4: Figure from Hartford et al. [2018]. The pink cube indicates an element in the output. It is a linear combination of the corresponding element in the input matrix (in dark blue), row sums (in green), column sums (in yellow), sums over the entire matrix (Pooling structure implied by the tied weights for matrices (left) and 3D tensors (right). The pink cube highlights one element of the output. It is calculated as a function of the corresponding element from the input (dark blue), pooled aggregations over the rows and columns of the input (green and yellow), and pooled aggregation over the whole input matrix (red). In the tensor case (right), we pool over all sub-tensors (orange and purple submatrices, green sub-vectors and red scalar). For clarity, the output connections are not shown in the tensor case

et al. [2019] discusses this exact case in more depth and applies the resulting permutation equivariant layers and architectures for predicting Premier League match outcomes using data on players, teams entities and venues.

2.3 Graph Neural Networks

Permutation symmetry is a central concern in graph learning. In supervised graph learning problems, we have a dataset of graphs that each come with an associated target value that we wish to predict: $\{(G_1, y_1), \ldots, (G_n, y_n)\}$, where the target value belongs to some output set $\mathcal{Y}$. Each graph $G_i = (V_i, E_i)$ is fully described by its set of vertices $V_i = \{v_1^{(i)}, v_2^{(i)}, \ldots, v_{n_i}^{(i)}\}$, and its set of edges $E_i \subseteq V_i \times V_i$. In many cases, the nodes of the graph may come with d-dimensional node features (ex: one hot atom embeddings for molecular graphs). Graphs are ubiquitous in the real world, as any data that contains any relational properties can be

encoded as a graph.

- A social network describing relationships between students at a college has edges between two people if they have ever sent an email to each other. The learning task here may be to predict potential new interactions between students.

- In the physical sciences, we often have molecular datasets, where we represent molecules as graphs. The nodes of the graph of a molecule are the constituent atoms and the edges denote bonds between atoms. Our learning task may be to predict a chemical property of a molecule.

- In e-commerce settings, we might have a database of items available for purchase by a website. Items in this database would be the nodes of the graph. Items have edges between them if they are commonly purchased together. The learning task may be to predict the best selling item in the future.

The central challenge in applying neural networks for graph learning problems is that we need our neural networks to take in variable sized inputs and they must also be permutation invariant. That is, if the vertices of an input graph are permuted/relabeled, the neural network must produce the same output. Formally, let f be our neural network. We need f to satisfy:

$$f(\{v_1, \ldots, v_n\}), E) = f(\{v_{\pi(i)}, \ldots, v_{\pi(n)}\}, E^{(\pi)}),$$

where $E^{(\pi)}$ denotes the edge set of the graph after vertices of G are relabeled with permutation π. The edge $(v_i, v_j) \in E$ if and only if $(v_{\pi(i)}, v_{\pi(j)}) \in E^{(\pi)}$ for all $\pi \in \mathbb{S}_n$. By now, there are standard ways to construct graph neural networks that can handle arbitrary sized inputs and exhibit the permutation invariance properties we desire. In graph neural networks, we start with input node features and iteratively construct updated node representation vectors based on the connectivity of the graph. The underlying principle that determines how nodes' hidden representations are updated is that its representation should be some combination

of its adjacent nodes' representations. While there has been quite a lot of research done on graph neural networks in recent years, Gilmer et al. [2017] stands out in articulating a general framework for how graph neural networks are generally constructed.

2.3.1 Message Passing Neural Networks

Gilmer et al. [2017] unifies the presentation of many existing graph neural networks by articulating them as message passing networks (MPNNs) where each node of the graph sends messages to its neighboring nodes. Each node aggregates the messages it receives to produce its own updated node representation. This message passing procedure occurs for T iterations until finally a readout function is applied over the nodes' hidden representations to produce a final graph embedding vector. This final graph representation can be used as a fixed sized input feature vector in a downstream multilayer perceptron.

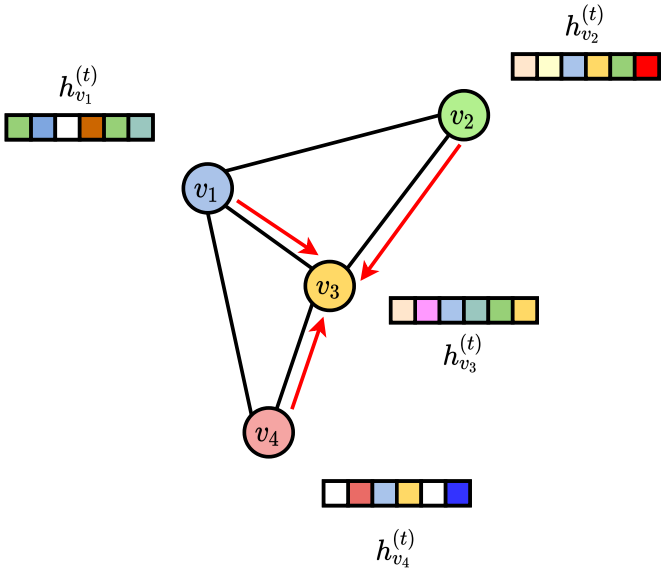


Figure 2.5: Each node v_i has an associated hidden representation $h_{v_i}^{(t)}$ during the tth round of message passing. In the graph above, vertex v_3 collects messages from its neighbors v_1, v_2 and v_4, to produce its updated node representation $h_{v_3}^{(t+1)}$.

For graph neural networks with T rounds of message passing, let M_t and U_t denote so-

called message functions and vertex update functions, which we can take to be multilayer perceptrons. Each node v has an associated hidden feature (or representation) vector, denoted h_v^t during the tth round of message passing. Each node's messages and hidden states are computed as follows during the $(t+1)-$th iteration:

$$m_v^{(t+1)} = \text{AGGREGATE}(\{M_t(h_v^{(t)}, h_w^{(t)}) \mid w \in \mathcal{N}(v)\})$$
$$h_v^{(t+1)} = U_t(h_v^{(t)}, m_v^{(t+1)})$$

The key step that makes the message passing is that the messages from the adjacent vertices are treated in a permutation invariant fashion—as in Deep Sets, the $m_v^{(t)}$ values are a symmetric combination of individual functions of the neighbors of v. In most graph neural network architectures, the AGGREGATE function is just summation over the messages.

After the T rounds of message passing, graph neural networks produce an output by applying a so-called "readout" function over the collection of final node representation vectors:

$$\widehat{y} = \text{READOUT}(\{h_v^{(T)} \mid v \in V\})$$

As with the AGGREGATE function, READOUT must also be a permutation invariant function. A common choice is to collapse the node features vectors with a permutation invariant function (e.g.: sum) and apply a multilayer perceptron:

$$\text{READOUT}(\{h_v^{(T)} \mid v \in V\}) = \text{MLP}(\sum_{v \in V} h_v^{(T)}).$$

In the presentation above, we did not mention edge features, but we can similarly construct hidden representations for the edges and modify the the AGGREGATE function to accept edge features as well.

There has been a subsequent explosion in graph neural network architectures, but most of

them operate by performing some version of the aforementioned message pass/aggregate/readout framework.

2.3.2 Convolutional Networks on Graphs for Learning Molecular Fingerprints

Duvenaud et al. [2015]'s graph convolutional network is introduced as a neural network approximation of circular fingerprints, where information about various substructures of the graph are hashed into a binary vector (the graph fingerprint). The proposed architecture produces the hidden representation of node v at each iteration by taking a sum of the previous layers' neighboring node representations, mixing the feature dimensions with a parameter matrix H_t and applying a nonlinearity σ(ReLU):

$$h_v^{(t+1)} = \sigma(H_t \sum_{w \in \mathcal{N}(v)} h_w^{(t)})$$

The output molecule representation $v_{fp} \in \mathbb{R}^d$ is computed by adding softmaxed node representations at the end of each iteration:

$$v_{fp} = \sum_{v \in V} \sum_{t=1}^{T} \text{softmax}(W_t h_v^{(t)})$$

where W_t's is a parameter matrix for the tth message passing iteration.

2.3.3 Gated Graph Sequence Networks

Li et al. [2015]'s Gated Graph Sequence Network takes node embeddings, propagates them according to the connectivity of the graph (sums node embeddings according to their local neighborhood), applies an individual shared neural network to each node embedding, and finally updates the nodes' hidden embeddings similar to gated recurrent units [Cho et al.,

2014] like fashion, where the new embedding is a function of the old embedding and the current iterations node messages.

$$
\begin{aligned}
a_v^{(t)} &= \sum_{w \in \mathcal{N}(v)} h_w^{(t)} \\
z_v^{(t)} &= \sigma(W_z^{(t)} a_v^{(t)} + U^{(t)} h_v^{(t-1)}) \\
r_v^{(t)} &= \sigma(W_r^{(t)} a_v^{(t)} + U^{(t)} h_v^{(t-1)}) \\
\tilde{h}_v^{(t)} &= \tanh(W a_v^{(t)} + U(r_v^{(t)} \odot h_v^{(t-1)})) \\
h_v^{(t)} &= (1 - z_v^t) h_v^{(t-1)} + z_v^{(t)} \tilde{h}_v^{(t)}
\end{aligned}
$$

The updates for a gated graph sequence network can be viewed as applying a recurrent network architecture on node representations. While the entire update procedure to produce $h_v^{(t+1)}$ from the previous iterations node representations is somewhat involved, the computation reduces to taking a nonlinear function of the $\sum_{w \in \mathcal{N}(v)} h_w^{(t)}$, and the previous layers hidden representation $h_v^{(t-1)}$.

2.3.4 GraphSage

Hamilton et al. [2017]'s GraphSage architecutre for learning on graphs uses the standard message pass, aggregate, readout framework. Their aggregate function takes the summation of node embeddings during the message passing step and applies a linearity and nonlinearity on the concatentation of the current iteration's message:

$$
\begin{aligned}
\tilde{h}_v^{(t)} &= \sum_{w \in \mathcal{N}(v)} h_w^{(t-1)} \\
h_v^{(t+1)} &= \sigma(W \cdot \text{concat}(h_v^{(t)}, \tilde{h}_v^{(t)}))
\end{aligned}
$$

where W is a parameter matrix used to take a linear combination of the feature indices of

the the previous hidden representation h_v^t and the aggregate of v's neighbors $\tilde{h}_v^t$.

2.3.5 Graph Attention Networks

Veličković et al. [2018] use an attention mechanism during the aggregation step of the message passing procedure. Instead of simply taking a summation over neighboring vertices, the new hidden representation of a given node is a weighted combination of the nodes in the neighborhood, where the weights are produced by attention over the neighborhood. The attention weights can be generated with standard dot product attention.

$$\alpha_{vw}^{(t)} = \frac{{h_v^{(t)}}^\top h_w^{(t)}}{\sum_{k \in \mathcal{N}(v)} {h_v^{(t)}}^\top h_k^{(t)}}$$

$$h_v^{(t)} = \sum_{w \in \mathcal{N}(v)} \alpha_{vw}^{(t)} h_w^{(t)}$$

2.3.6 Covariant Composition Networks

Another interpretation for standard graph neural network layers, is that the aggregation step at each graph neural network layer is effectively applying a Deep Sets styled permutation invariant function on the set of node messages. Similarly, the readout function is also applying a Deep Sets styled architecture on the set of all final node representation vectors. Under this interpretation, it is natural to wonder if we might be able to construct more complicated or expressive aggregation and readout functions. Can we use higher order permutable tensors in the message passing step or otherwise do more than just sum to aggregate?

Covariant Compositional Networks [Kondor et al., 2018b] formally discusses graph neural networks that can incorporate higher order information between vertices. In the previous section on graph neural networks, a node v of a graph in an MPNN passes messages $m_v^{(t)} \in \mathbb{R}^d$ to its adjacent nodes during the t'th round of message passing. CCN introduces the concept of

higher order message passing, where nodes pass around arbitrary ordered permutable tensors as their messages. In the simplest case, a node's message $m_v^{(t)} \in \mathbb{R}^{|\mathcal{N}(v)| \times d}$ is a 1st order permutable tensor. We can have a node's message $m_v^{(t)}$ be a higher ordered permutable tensor as well: $m_v^{(t)} \in \mathbb{R}^{|\mathcal{N}(v)|^k \times d}$, for $k = 2, 3$, etc. In practice, we usually have $k \leq 4$. The symmetric pooling operations can be postponed until we need a lower ordered tensor elsewhere in the architecture.

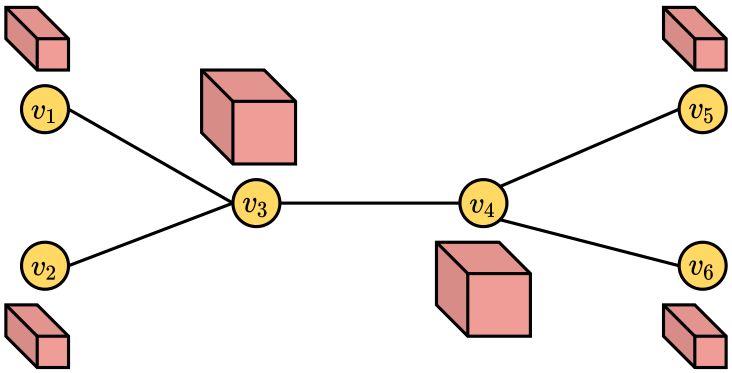


Figure 2.6: A node's hidden representation in a CCN [Kondor et al., 2018b] can be a higher order permutable tensor. In the figure above, each node passes a 2^{nd} order permutable tensor to its neighbors during the message passing step of a CCN. The depth of the visualized node representation tensors is the feature dimension.

Kondor et al. [2018b] also formally describes the operations that can be applied to permutable tensors that maintain the permutable structure of the tensor.

Theorem 3. *Let $A \in \mathbb{R}^{n^{k_1} \times d}$ be a k_1th order permutable tensor with d channels, and B be a k_2th order permutable tensor. The following tensor operations on permutable tensors are permutation equivariant:*

- *tensor products between permutable tensors*
- *elementwise products over entity indices*
- *summations over entity indices (contractions over indices)*

Operations on permutable tensors that result in a permutable tensor are important because they provide flexible toolkit of operations that respect permutation equivariance. The key innovation of CCNs is that these tensor operations are permutation equivariant and thus we can freely use any of these tensor operations in sequence with any other permutation equivariant operations and still maintain the desired equivariance property.

Kondor et al. [2018a] also gives a generic architecture for a permutation equivariant higher order message passing architecture. Each node v is associated with a hidden k'th order permutable tensor $h_v^{(t)}$ at step t of the message passing procedure, which proceeds as follows:

- Each node's hidden feature vector is updated with a row-wise multilayer perceptron or linear layer: $\tilde{h}_v^{(t)} \leftarrow MLP(h_v^{(t)})$
- Each node v receives messages $h_w^{(t)}$ from each vertex w in its receptive field (ex: neighboring vertices)
- This collection of kth order tensors: $\{h_w^{(t)} \mid w \in \mathcal{N}(v)\}$ can be combined to form a $(k+1)$'th order tensor and (optionally) tensor producted with a local adjacency matrix to form a $(k+3)$'th order tensor
- The higher ordered tensor from previous step can be cast back down to a kth order permutable tensor by a contraction over two entity indices

As in standard graph neural networks, we apply some readout function that combines the k'th order representations of each node in a permutation invariant fashion to produce a scalar output.

Kondor et al. [2018b] was the first work to consider higher order message passing, and also the first work to consider the set of valid equivariant operations that could be applied on higher order tensors. In some ways, our work on permutation equivariant layers for higher interactions is a natural spiritual successor to CCNs as it applies many the same

guiding principles for designing permutation equivariant neural networks notably the use of tensor/outer products to construct higher order permutable tensors from lower order permutable tensors.

2.3.7 Invariant and Equivariant Graph Networks

Our work on higher order layers for permutation equivariant networks builds upon *Invariant and Equivariant Graph Networks* [Maron et al., 2018], which derives the precise parameter sharing scheme in a linear permutation equivariant layer that maps a k_1'th order permutable tensors to a k_2'th order permutable tensors.

Following the notation used by Hartford et al. [2018], let $L \in \mathbb{R}^{n^k \times n^k}$, and $f_L : \mathbb{R}^{n^k} \to \mathbb{R}^{n^k}$ be the linear function where $f_L(X) = L(X)$. The seminal finding of Maron et al. [2018] is that f_L is a linear permutation equivariant function if and only if: $\pi \cdot L = L$ for all $\pi \in \mathbb{S}_n$. Recall that permutations act on the weight matrix L by permuting the indices of L:

$$[\pi \cdot L]_{i_1,\ldots,i_{2k}} = L_{\pi^{-1}(i_1),\ldots,\pi^{-1}(i_{2k})}.$$

To properly present Maron et al. [2018]'s parameter sharing scheme for the weight matrix L, we first introduce the concept of equivalence classes of indices of permutable tensors. L can be indexed by a multi-index in $\{1, 2, \ldots, n\}^{2k}$.

$$[Lx]_{i_1,i_2,\ldots,i_k} = \sum_{(j_1,\ldots,j_k)\in[n]^k} L_{(i_1,\ldots,i_k,j_1,\ldots,j_k)} X_{j_1,\ldots,j_k}$$

In subsequent sections, we may also index L by a pair of two multi-indices in $[n]^k$ for notational clarity to explicitly indicate the input and output indices in the summation.

Two multi-indices $\mathbf{a}, \mathbf{b}$ of L are in the same equivalence class if they share the same equality pattern. That is, $a_i = a_j$ if and only if $b_i = b_j$. The definition of this equality pattern is slightly opaque so we present a few examples to better illustrate the idea.

Example 9. *For 2nd order permutable tensors $T_2 \in \mathbb{R}^{n \times n}$, we have two equivalence classes of indices of T_2: the diagonal entries: $\{(i,i) \mid i \in [n]\}$ and the off-diagonal entries: $\{(i,j) \mid i \in [n], j \in [n], i \neq j\}$.*

Perhaps more importantly, the weight matrix underlying the linear permutation equivariant function has the following parameter sharing scheme: indices of L that have the same index equality pattern share the same value.

Maron et al. [2018] shows that the number of free parameters between a k_1'th order permutable tensor and a k_2'th order permutable tensor (assuming both have a feature dimension of 1) is exactly $B(k_1 + k_2)$, where $B(n)$ denotes the nth Bell number. The number of index equality patterns is equal to the number of partitions of $k_1 + k_2$ elements.

Definition 2.3.1 (Partition)**.** *A partition of a set is a grouping of its elements into non-empty subsets such that every element of the original set is included in exactly one subset.*

Definition 2.3.2 (Bell Number)**.** *The n'th Bell number, denoted $B(n)$, counts the number of different ways to partition the set $\{1, 2, \ldots, n\}$.*

The following notation is used to index elements of a k'th order tensor according to partitions of k. Given a k'th order tensor $M \in \mathbb{R}^{n^k}$, and a partition $\mathcal{P}$ of $\{1, \ldots, k\}$, we use the notation $M_{\mathcal{P}}$ to denote the set of indices of M that have an equality pattern corresponding to $\mathcal{P}$: indices in the same subset have the same value and indices in different subsets have different values.

Example 10. *An adjacency matrix A is a second order tensor. For $k = 2$, we only have two partitions of $\{1, 2\}$: $\mathcal{P}_1 = \big\{\{1, 2\}\big\}$ and $\mathcal{P}_2 = \big\{\{1\}, \{2\}\big\}$. The entries of A corresponding to partition $\{\{1, 2\}\}$ are the diagonal entries: $A_{\mathcal{P}_1} = \{A_{ij} \mid i = j\}$. The entries of A that correspond to partition $\big\{\{1\}, \{2\}\big\}$ are the off-diagonal entries: $A_{\mathcal{P}_2} = \{A_{ij} \mid i \neq j\}$.*

Example 11. *Let $M \in \mathbb{R}^{n \times n \times n}$ be a third order permutable tensor. There are five partitions*

of $\{1, 2, 3\}$*, which are:*

$$\big\{\{1, 2, 3\}\big\}, \big\{\{1, 2\}, \{3\}\big\}, \big\{\{1, 3\}, \{2\}\big\}, \big\{\{2, 3\}, \{1\}\big\}, \big\{\{1\}, \{2\}, \{3\}\big\}.$$

Therefore, we have five equivalence classes of entries of M*, each corresponding to one of the partitions:*

- $\big\{\{1, 2, 3\}\big\}$ *corresponds to the entries:* $\{M_{iii} \mid i \in \{1, \ldots, n\}\}$
- $\big\{\{1, 2\}, \{3\}\}\big\}$ *corresponds to the entries:* $\{M_{iij} \mid i, j \in \{1, 2, \ldots, n\}, i \neq j\}$
- $\big\{\{1, 3\}, \{2\}\}\big\}$ *corresponds to the entries:* $\{M_{iji} \mid i, j \in \{1, 2, \ldots, n\}, i \neq j\}$
- $\big\{\{2, 3\}, \{1\}\}\big\}$ *corresponds to the entries:* $\{M_{jii} \mid i, j \in \{1, 2, \ldots, n\}, i \neq j\}$
- $\big\{\{1\}, \{2\}, \{3\}\big\}$ *corresponds to the entries:* $\{M_{ijk} \mid i, j, k \in \{1, 2, \ldots, n\}, i \neq j \neq k\}$

Maron et al. [2018] is significant as it explicitly describes the entire parameter sharing scheme that is necessary and sufficient for linear permutation equivariant layers between arbitrarily ordered permutable tensors. They also demonstrated the effectiveness of second order equivariant layers in a graph neural network architecture on various graph datasets such as QM9.

2.4 Attention

Finally, we would be remiss not to mention the popular attention mechanism. We discuss attention last even though it preceded Deep Sets and PointNet because it was conceived entirely separately from the rest of the literature on permutation equivariance. The fact that it posseses the permutation symmetry preserving properties we are intersted in almost seems to be an afterthought. Bahdanau et al. [2015] first introduced the attention mechanism for sequence to sequence modeling. Previously, recurrent neural network layers[Rumelhart et al.,

1985], long short term memory units[Hochreiter and Schmidhuber, 1997], and gated recurrent units [Cho et al., 2014] were the standard building blocks for modeling text and sequence data. Vaswani et al. [2017] famously demonstrated that attention layers were sufficient for sequence learning tasks; their Transformer architecture, which interleaved multiple attention layers and row-wise feed forward layers with residual connections, outperformed recurrent neural network based architectures.

An attention layer takes in three input matrices, often referred to as a query matrix $Q \in \mathbb{R}^{n \times d_k}$, key (or context) matrix $K \in \mathbb{R}^{n \times d_k}$, and a value matrix $V \in \mathbb{R}^{n \times d_v}$. The output is a linear combination of the rows of V, where the coefficients are determined by the corresponding query and key vectors' "compatibility." Formally, the attention layer is defined follows: $\text{Att} : \mathbb{R}^{n \times d_k} \times \mathbb{R}^{n \times d_k} \times \mathbb{R}^{n \times d_v} \to \mathbb{R}^{n \times d_v}$.

$$\text{Att}(Q, K, V) = \text{softmax}(\frac{QK^\top}{\sqrt{d_k}})V$$

If we inspect the (i, j) entries of the $\text{softmax}(\frac{QK^\top}{\sqrt{d_k}})$ term, we can see what these coefficients are precisely:

$$\alpha_{ij} = \left[\text{softmax}\frac{QK^\top}{\sqrt{d_k}}\right]_{ij} = \frac{Q_i^\top K_j}{\sum_{l=1}^{n} Q_i^\top K_l}$$

These α_{ij} values are also commonly referred to as attention weights or attention coefficients. There are other forms of constructing these Inspecting individual indices of the output of the attention layer in terms of these attention coefficients, for the i row of $\text{Att}(Q, K, V)$, we have:

$$\Big[\text{Att}(Q, K, V)\Big]_i = \sum_{j=1}^{n} \alpha_{ij} V_j,$$

where V_j denotes the jth row of V.

2.4.1 Self Attention

Attention layers were first used in the context of machine translation [Cho et al., 2014], so the query and key matrices corresponded to embeddings of the input sequence of words in a source language and embeddings of the output sequence of words in the target language respectively. This formulation allows the output translated words to capture the relationship between the pre and post translated words. On the other hand, we may also have query and key matrices originating from the same input sequence, leading to what is commonly referred to as **self attention**.

A self attention layer takes the matrix $X \in \mathbb{R}^{n \times d}$ as input. It has learnable parameters $W_Q, W_K \in \mathbb{R}^{d \times d_k}, W_V \in \mathbb{R}^{d \times d_v}$ which produce the query, key and value matrices.

$$\text{SelfAtt}(X; W_Q, W_K, W_V) = \text{Att}(XW_Q, XW_K, XW_V).$$

While attention layers were not developed with permutation invariance or equivariance in mind, self attention is in fact a permutation equivariant operation. Notice that the linear mixings applied by W_q and W_k are not affected by a permutation action. Consider an arbitrary permutation $\pi \in \mathbb{S}_n$ and an any index i. Let

$$\begin{aligned} Q^{(\pi)} &= (\pi \cdot X)W_q \\ K^{(\pi)} &= (\pi \cdot X)W_k \\ V^{(\pi)} &= (\pi \cdot X)W_v \end{aligned}$$

denote the query, key and value matrices after the permutation action by π respectively. Inspecting the ith index of the output of the self attention layer after permuting the rows of

Q, K, V, we see that we have:

$$\begin{aligned}\Big[Att(Q^{(\pi)}, K^{(\pi)}, V^{(\pi)})\Big]_i &= \left(\frac{Q_{\pi^{-1}(i)}^{\top} K_{\pi^{-1}(j)}}{\sqrt{d_k} \sum_{l=1}^{n} Q_{\pi^{-1}(i)} V_l}\right) V_{\pi^{-1}(i)} \\ &= \Big[\text{Att}(Q, K, V)\Big]_{\pi^{-1}(i)}\end{aligned}$$

Thus, self-attention is permutation equivariant. Another perspective on this comes from noticing that the $n \times n$ matrix of attention coefficients, $\text{softmax}\left(\frac{QK^{\top}}{\sqrt{d_k}}\right)$, is a second order permutable tensor (with implicit feature dimension 1), and V is a first order permutable tensor. The matrix multiplication of the two permutable tensors will result in a permutable tensor, giving us the desired permutation equivariance as well. Finally, self attention can be understood as a generalization of the Deep Sets layer. If we set all of the α_{ij} attention coefficients to 1, then the output of the resulting attention layer would simply be a summation over the rows of V. Combining the attention layer with a residual connection from the input, we get exactly the result of a Deep Sets permutation equivariant layer.

Given that attention layers can be used anywhere where we have a summation over entities, it is not surprising that attention layers and transformer styled architectures have been used in graph models such as the Graph Attention Network[Veličković et al., 2018], where the aggregation step of the message passing layer uses attention coefficients. Attention layers have also been applied in the Set Transformer Lee et al. [2019a], SE(3)-Transformer[Fuchs et al., 2020] and LieTransformer Hutchinson et al. [2021] architectures for learning tasks on data exhibiting rotational, translational, and permutation symmetries.

CHAPTER 3

HIGHER ORDER PERMUTATION EQUIVARIANT LAYERS

Prior work on permutation equivariant layers provided the groundwork for conceptualizing how a linear map between permutable tensors might look like. Maron et al. [2018] in particular, gives sufficient conditions on how various entries in the matrix underlying a permutation equivariant linear map must be tied together. We would like a general prescription for how to construct these permutation equivariant linear layers between arbitrary k_1'th order permutable tensors to k_2'th order permutable tensors in a practical way. We begin by considering small values of k_1 and k_2.

3.1 Deriving Equivariant Layers

For small values of k_1 and k_2, it is possible to derive the form of linear equivariant layer mapping k_1'th to k_2'th order permutable tensors just by reasoning about the set of symmetric linear operations that are possible.

3.1.1 Case I: 1 to 1 layer

There are two trivial ways to construct linear permutation equivariant mappings from one first order tensor $X \in \mathbb{R}^n$ to another, $Y \in \mathbb{R}^n$:

1. The identity map (multiplied by a scalar):

$$Y_i \leftarrow \lambda_1 X_i$$

2. The averaging map (multiplied by a scalar):

$$Y_i \leftarrow \lambda_2 \sum_j X_j$$

Zaheer et al. [2017] proves that in fact these are the *only* two possibilities. Therefore, the most general first order permutation equivariant layer in matrix form is

$$Y = \lambda_1 X + \lambda_2 \mathbf{1}\mathbf{1}^\top X,$$

giving us only two learnable parameters λ_1 and λ_2.

3.1.2 Case II: 1 to 2 layer

Now consider a linear permutation equivariant function that maps a first order permutable tensor $X \in \mathbb{R}^n$ to a second order tensor $Y \in \mathbb{R}^{n \times n}$. Let us consider how we might construct entry Y_{ij}. We have two types of entries to be updated: the diagonal and off-diagonal entries. The off-diagonal entries Y_{ij} can be set to a linear combination of X_i, X_j, and the sum of all the X_k's. The diagonal entries Y_{ii} by themselves form a first order permutable tensor, so following Case I3.1.1, they can be set to a linear combination of X_i and $\sum_k X_k$:

$$Y_{ij} \leftarrow \lambda_1 X_i + \lambda_2 X_j + \lambda_3 (\sum_k X_k)$$
$$Y_{ii} \leftarrow \lambda_4 X_i + \lambda_5 (\sum_k X_k)$$

Thus, Y is a linear combination of five second order permutable tensors:

1. a tensor with X broadcast over the rows of the output tensor
2. a tensor with X broadcast over the columns of the output tensor
3. a tensor with the global sum of X tiled over all entries
4. a tensor with X embedded in the diagonal
5. a tensor with the global sum of of X tiled over the diagonal

which gives us a total of 5 learnable parameters.

3.1.3 Case III: 2 to 1 layer

Now suppose our input tensor is $X \in \mathbb{R}^{n \times n}$ and our output tensor is $Y \in \mathbb{R}^n$. Consider the elements of X that affect the i'th element. The i'th output Y_i can involve diagonal entries X_{ii}, the i'th row sum, the i'th column sum, and the global sum of all of X. Lastly, we can also construct a symmetric element by taking the sum of the diagonal elements of X.

$$Y_i \leftarrow \lambda_1 X_{ii} + \lambda_2 \sum_j X_{ij} + \lambda_3 \sum_j X_{ji} + \lambda_4 \sum_{ij} X_{ij} + \lambda_5 \sum_i X_{ii}$$

Similar to the previous case, we can see that Y is a linear combination of five 1st order tensors:

- X summed over its row index
- X summed over its column index
- the diagonal of X
- the global sum of X tiled into a 1st order tensor
- the sum of the diagonal entries of X tiled into a 1st order tensor

again giving us 5 learnable parameters.

3.1.4 Case IV: 2 to 2 Layer

In a $2 \rightarrow 2$ equivariant layer, our input tensor is $X \in \mathbb{R}^{n \times n}$ and our output tensor is $Y \in \mathbb{R}^{n \times n}$. The expected number of parameters for an equivariant layer mapping from a 2nd order to 2nd order is $B(2+2) = 15$, according to Maron et al. [2018]. Figuring out all the symmetric ways of aggregating terms in X that may affect entries Y_{ij} starts to get a bit cumbersome. We provide a full enumeration of all the updates to the ouptut 2nd order tensors in Table 3.1. One thing to notice is that there are a few intermediate tensors that

are reused in multiple operations get re-used in multiple operations (ex: row sum, column sum, diagonal of the original X tensor) .

	Partition	Pattern	Update	Description
1	$\{1,2,3,4\}$	$(i_1, i_1, \mid i_1, i_1)$	$[T_2]_{ii} \leftarrow \lambda_1 [T_1]_{ii}$	Diag of T_1 sent to diag of T_2
2	$\{1\},\{2,3,4\}$	$(i_1, i_2, \mid i_2, i_2)$	$[T_2]_{ij} \leftarrow \lambda_2 [T_1]_{jj}$	Diag of T_1 sent to rows of T_2
3	$\{2\},\{1,3,4\}$	$(i_1, i_2, \mid i_1, i_1)$	$[T_2]_{ij} \leftarrow \lambda_3 [T_1]_{ii}$	Diag of T_1 sent to cols of T_2
4	$\{3\},\{1,2,4\}$	$(i_1, i_1, \mid i_2, i_1)$	$[T_2]_{ii} \leftarrow \lambda_4 \sum_j [T_1]_{ji}$	Row sum of T_1 sent to diag of T_2
5	$\{4\},\{1,2,3\}$	$(i_1, i_1, \mid i_1, i_2)$	$[T_2]_{ii} \leftarrow \lambda_5 \sum_j [T_1]_{ij}$	Col sum of T_1 sent to diag of T_2
6	$\{1,2\},\{3,4\}$	$(i_1, i_2, \mid i_3, i_4)$	$[T_2]_{ii} \leftarrow \lambda_6 \sum_j [T_1]_{jj}$	Diag sum of T_1 sent to diag of T_2
7	$\{2,3\},\{1,4\}$	$(i_1, i_2, \mid i_2, i_1)$	$[T_2]_{ij} \leftarrow \lambda_7 [T_1]_{ji}$	Transpose of T_1 sent to T_2
8	$\{1,3\},\{2,4\}$	$(i_1, i_2, \mid i_1, i_2)$	$[T_2]_{ij} \leftarrow \lambda_8 [T_1]_{ij}$	T_1 sent to T_2
9	$\{1\},\{2\},\{3,4\}$	$(i_1, i_2, \mid i_3, i_3)$	$[T_2]_{ij} \leftarrow \lambda_9 \sum_k [T_1]_{kk}$	Diag sum of T_1 sent to T_2
10	$\{1\},\{3\},\{2,4\}$	$(i_1, i_2, \mid i_3, i_2)$	$[T_2]_{ij} \leftarrow \lambda_{10} \sum_k [T_1]_{kj}$	Row sum of T_1 sent to cols of T_2
11	$\{1\},\{4\},\{2,3\}$	$(i_1, i_2, \mid i_2, i_3)$	$[T_2]_{ij} \leftarrow \lambda_{11} \sum_k [T_1]_{jk}$	Col sum of T_1 sent to cols of T_2
12	$\{3\},\{4\},\{1,2\}$	$(i_1, i_1, \mid i_3, i_4)$	$[T_2]_{ii} \leftarrow \lambda_{12} \sum_{k,l} [T_1]_{kl}$	Sum of all of T_1 sent to diag of T_2
13	$\{2\},\{4\},\{1,3\}$	$(i_1, i_2, \mid i_1, i_3)$	$[T_2]_{ij} \leftarrow \lambda_{13} \sum_k [T_1]_{ik}$	Col sum of T_1 sent to rows of T_2
14	$\{2\},\{3\},\{1,4\}$	$(i_1, i_2, \mid i_3, i_1)$	$[T_2]_{ij} \leftarrow \lambda_{14} \sum_k [T_1]_{ki}$	Row sum of T_1 sent to rows of T_2
15	$\{1\},\{2\},\{3\},\{4\}$	$(i_1, i_2, \mid i_3, i_4)$	$[T_2]_{ij} \leftarrow \lambda_{15} \sum_{kl} [T_1]_{kl}$	Sum of all of T_1 sent to T_2

Table 3.1: Operations to construct the broadcast tensors in a 2 to 2 layer

While it is still somewhat manageable to enumerate all possible operations in the $2 \to 2$ equivariant layer, it quickly becomes untenable to try to figure out the necessary operations and tensors involved in linear permutation equivariant layers involving anything higher than 2nd order permutable tensors. We would like to have a systematic way of describing the linear combination of tensors that can be a part of the output of the layer.

3.2 Deriving Equivariant Layers From Partitions

We can also derive the full set of linear permutation equivariant transformations using the line of reasoning set forth by Maron et al. [2018]. First, we recall their findings on permutation equivariant layers.

Theorem 4 (Maron et al. [2018])**.** *A matrix $M \in \mathbb{R}^{n^{k_2} \times n^{k_1}}$ is a permutation equivariant linear map from $\mathbb{R}^{n^{k_1}} \to \mathbb{R}^{n^{k_2}}$ if and only if M is invariant to its permutation action. If we*

view M as a (k_1+k_2)-th dimensional array. The permutation action on M is given by:

$$[\pi \cdot M]_{i_1 \dots i_{k_1+k_2}} = M_{\pi^{-1}(i_1)\dots\pi^{-1}(i_{k_1+k_2})}.$$

for any $\pi \in \mathbb{S}_n$. For any two indices of M written as (k_1+k_2)-tuples: $I = (i_1, \dots, i_{k_1+k_2})$ and $J = (j_1, \dots, j_{k_1+k_2})$ where each of the indices of I and J are in $\{1, \dots, n\}$, if $\pi \cdot I = J$, then $M_I = M_J$. In other words, the values of M are the same along indices with the same partition pattern. Therefore, M has exactly $B(k_1+k_2)$ degrees of freedom, each corresponding to a different partition pattern.

This theorem implies that if we are interested in understanding the constituent parts of a permutation equivariant linear map M, it is necessary to inspect its indices corresponding to the same partition pattern in M for every partition $\mathcal{P}$ of $\{1, 2, \dots, k_1+k_2\}$.

To better understand the implications of M being constant on each equivalence class of indices, let T_1 and T_2 be 3rd order tensors ($k_1 = k_2 = 3$) where $T_2 = MT_1$ for some linear permutation equivariant linear map $M \in \mathbb{R}^{n^{k_2} \times n^{k_1}}$. We will also use the notation $M_{[\mathcal{P}]} \in \mathbb{R}$ to denote the shared scalar value in the entries of M corresponding to partition $\mathcal{P}$.

Let $\mathcal{P} = \big\{\{1\}, \{2,3,4\}, \{5,6\}\big\}$ be our running example partition/equivalence class. We associate a separate index variable $i_1, i_2, \dots i_{|\mathcal{P}|}$ to each part of $\mathcal{P}$. For this partition, if we list which index variable each of the original indices is associated with, we have $(i_1, i_2, i_2, i_2, i_3, i_3)$. We will also draw a vertical line as a visual aid to demark the boundary between "output" and "input" index variables: $(\underbrace{i_1, i_2, i_2}_{k_2} \mid \underbrace{i_2, i_3, i_3}_{k_1})$ The part of our permutation equivariant map ϕ corresponding to partition $\mathcal{P}$ is then:

$$[T_2]_{i_1,i_2,i_2} \leftarrow \sum_{i_3} M_{i_1,i_2,i_2|i_2,i_3,i_3} [T_1]_{i_2,i_3,i_3} \tag{3.1}$$

$$[T_2]_{i_1,i_2,i_2} \leftarrow M_{[\mathcal{P}]} \cdot \sum_{i_3} [T_1]_{i_2,i_3,i_3} \tag{3.2}$$

Recall, that the elements of the weight matrix underlying a permutation equivariant layer must be constant on indices with the same partition pattern, hence $M_{i_1,i_2,i_2|i_2,i_3,i_3} = M_{[\mathcal{P}]}$ for all distinct i_1, i_2, i_3. So we can pull the constant $M_{[\mathcal{P}]}$ out of the summation in Equation (3.2).

The full linear map can be computed by evaluating an update analogous to Equation 3.1) for every partition $\mathcal{P}$ of $(k_1 + k_2)$. The advantage of breaking down the computation of entries of T_2 in the manner shown by Equation (3.2) is that we can systematically isolate the operations that are applied only on T_1 and operations that need to be applied to T_2. In Equation 3.2 above, the operation applied to T_1 amounted to a summation over the diagonal of its last two indices.

Returning to our visual aid to denote the first k_2 indices and last k_1 indices, we make a distinction between variables based on whether they are input or output variables (which side of the dividing line they reside on).

1. Variables that only appear on the right hand side of the line serve as dummy indicies for summation. We call these **summation variables**.

2. Variables that appear on the left side of the line serve as "free" variables in the output tensor which determine how the result is broadcast into T_2. If a given index appears more than once, then the result is pushed to the corresponding diagonal slice of T_2. These variables are called **broadcast variables**.

3. Finally, there are variables that appear on both sides of the dividing line such as i_2 in our example above. These tie together the input and output side, therefore we call them **transfer variables**.

3.3 Organizing the computation

The summation appearing in Eqn. (3.1) is shared across all partitions $\mathcal{P}$ that partition $k_2+1, ..., k_2+k_1$ in the same way. We can first compute all possible summations and then pair them with all possible left hand sides (of the dividing line) and broadcast the result into T_2. There are three key quantities for organizing this computation:

- n_s, the number of indices on the right of the dividing line that are associated with summation variables,
- n_b, the number of indices on the left which are associated with the broadcast variables,
- κ, the number of transfer variables.

3.3.1 Summation Tensors

There are $\binom{k_1}{n_s}$ ways of choosing summation indices. For each choice, there are $B(n_s)$ ways of partitioning the n_s summation indices into distinct summation variables. If two summation indices are grouped together, then the summation occurs on the "diagonal". Therefore, the total number of distinct expressions like Equation (3.1) must be:

$$\sum_{n_s=0}^{k_1} \binom{k_1}{n_s} B(n_s)$$

The complexity of each of these sums is $O(n^{n_s})$. Each of these sums has $k_1 - n_s$ unbound indices, so the summation results in a tensor of order $k_1 - n_s$.

Example 12. *For $k_1 = 3$, we have: $\binom{3}{0}B(0)+\binom{3}{1}B(1)+\binom{3}{2}B(2)+\binom{3}{3}B(3) = 1+3+6+5 = 15$ different tensors that appear in the case of $k_1 = 3$.*

The crucial takeaway here is that summations over various indices can be reused for various parts of the equivariant layer if they share the same summation variables and partition

pattern. Recall in Table 3.1, we saw that row sums, column sums, and the diagonal of T_1 were reused in multiple broadcast operations.

3.3.2 Transfer Operations

The κ transfer variables are assigned to the $k_1 - n_s$ indices of the summation tensors, but we have to introduce yet another partition index $\mathcal{P}_t$, since multiple indices of the summation tensor may be tied to the same transfer variable. In addition, we need to account for the $\kappa!$ possible permutations of the transfer variables. The total number of such transfer tensors is[1]

$$B(k_1) + \sum_{\kappa=1}^{k_1} \kappa! \sum_{n_s=0}^{k_1} S(k_1 - n_s, \kappa) \binom{k_1}{n_s} B(n_s)$$

where $S(k_1 - n_s, \kappa)$ denotes the number of ways to split $k_1 - n_s$ among κ non-empty sets.

The transfer variables in Equation (3.1) tell us where to send the result of our intermediate summation operations.

Definition 3.3.1 (Stirling Numbers of the second kind)**.** *Stirling numbers of the second kind, denoted $S(n, k)$, count the number of ways to partition a set of n labeled elements into k non-empty subsets.*

Example 13. *Consider the partition $\mathcal{P} = \{\{1, 4\}, \{2, 5\}, \{3, 6\}\}$ for a 3rd order to 3rd order equivariant layer. The number of transfer variables is $\kappa = 3$ and the output tensor associated with this partition is: $[T_2]_{i_1,i_2,i_3} \leftarrow M_{[\mathcal{P}]}[T_1]_{i_1,i_2,i_3}$. We could also have the transfer operation be slightly different according to the 3! ways we could have allotted the transfer variables from the k_2 side among the k_1 side's transfer variables, which would give us the following*

1. $S(n, k)$ denotes a Stirling number of the 2nd kind

alternative entry contributions to T_2:

$$
\begin{aligned}
[T_2]_{i_1,i_2,i_3} &\leftarrow \lambda_1 [T_1]_{i_1,i_2,i_3} \\
[T_2]_{i_1,i_3,i_2} &\leftarrow \lambda_2 [T_1]_{i_1,i_2,i_3} \\
[T_2]_{i_2,i_1,i_3} &\leftarrow \lambda_3 [T_1]_{i_1,i_2,i_3} \\
[T_2]_{i_2,i_3,i_1} &\leftarrow \lambda_4 [T_1]_{i_1,i_2,i_3} \\
[T_2]_{i_3,i_1,i_2} &\leftarrow \lambda_5 [T_1]_{i_1,i_2,i_3} \\
[T_2]_{i_3,i_2,i_1} &\leftarrow \lambda_6 [T_1]_{i_1,i_2,i_3}.
\end{aligned}
$$

3.3.3 Broadcast Tensors

Finally, for each transfer tensor we need to consider how many different ways it can be applied to the output tensor. For any value of κ, n_b must be in the range $0 \leq n_b \leq k_2 - \kappa$ and for each value of n_b we can choose which indices become broadcast versus transfer variables as well as how the broadcast and transfer indices are partitioned into variables amongst themselves. So the total number of ways of broadcasting such a tensor is: $\sum_{n_b=0}^{k_2} S(k_2 - n_b, \kappa) \binom{k_2}{n_b} B(n_b)$. For every possible way of constructing a transfer tensor, we have the above number of ways to mold it into a broadcast tensor. Thus, the total number of broadcasting operations is: $B(k_1)B(k_2) + \sum_{\kappa=1}^{\min(k_1,k_2)} \kappa! \big[\sum_{n_s=0}^{k_1} S(k_1 - n_s) \binom{k_1}{n_s} B(n_s) \big] \big[\sum_{n_b=0}^{k_2} S(k_2 - n_b) \binom{k_2}{n_b} B(n_b) \big]$. Theorem 5 shows that this convoluted expression is in fact equal to the Bell number $B(k_1 + k_2)$, which is the expected number of operands in a $\mathbb{R}^{n^{k_1}} \to \mathbb{R}^{n^{k_2}}$ permutation equivariant linear layer as derived by Maron et al. [2018].

In practice, the transfer and broadcasting parts of this framework can be implemented in GPU accelerated numerical libraries such as PyTorch [Paszke et al., 2019] with various primitive tensor operations.

Theorem 5.

$$\begin{aligned} B(k_1+k_2) =& B(k_1)B(k_2) + \sum_{\kappa=1}^{\min(k_1,k_2)} \kappa! \\ &\times \Big[\sum_{n_s=0}^{k_1} S(k_1-n_s,\kappa)\binom{k_1}{n_s} B(n_s)\Big] \\ &\times \Big[\sum_{n_b=0}^{k_2} S(k_2-n_b,\kappa)\binom{k_2}{n_b} B(n_b)\Big] \end{aligned}$$

Proof. Suppose we have $k_2 + k_1$ balls arranged from left to right. By definition of the Bell numbers, the number of ways to partition these balls into non-empty subsets is $B(k_1 + k_2)$.

We can also count the number of partitions of these k_2+k_1 balls by counting the number of ways we can construct partitions with κ subsets containing balls spanning the first k_2 balls and the last k_1 for $\kappa = 0, 1, ...$ balls. Let n_b denote the number of balls assigned to subsets of the partition that contained within the first k_2 balls, and n_s denote the number of balls assigned to subsets contained in the last k_1 balls.

Case 1: $\kappa = 0$. There are $B(k_2)$ ways to partition the first k_2 balls and likewise $B(k_1)$ ways to partition the last k_1 balls, giving us a total of $B(k_1)B(k_2)$ total ways to partition the balls such that no subset of the partition spans the two sides.

Case 2: $\kappa > 0$. For this case we consider n_s, n_b, the number of balls on the two sides that belong to sets that do not cross the divide. For each possible value of n_s, there are $\binom{k_1}{n_s}$ ways to pick the n_s balls, and $B(n_s)$ ways to partition these balls. In the remaining $k_1 - n_s$ balls on this side of the divide, we can split them among the κ subsets that cross the divide in $S(k_1 - n_s, \kappa)$ ways by definition of the Stirling numbers of the second kind. This gives us the expression $\sum_{n_s=0}^{k_1} S(k_1 - n_s, \kappa)\binom{k_1}{n_s} B(n_s)$. By symmetry, the expression for the number of ways to pick the n_b balls in subsets lying entirely on the k_2 side of the divide must be $\sum_{n_b=0}^{k_2} S(k_2 - n_s, \kappa)\binom{k_2}{n_b} B(n_b)$. For each possible value of κ, we can permute the ordering of these subsets $\kappa!$ ways. Putting everything together, we have the expression

$\sum_{\kappa=1}^{\min(k_1,k_2)} \kappa! \big[\sum_{n_s=0}^{k_1} S(k_1 - n_s, \kappa) \binom{k_1}{n_s} B(n_s) \big] \big[\sum_{n_b=0}^{k_2} S(k_2 - n_b, \kappa) \binom{k_2}{n_b} B(n_b) \big]$ as desired. □

Theorem 5 tells us that the number of ways of constructing broadcast tensors according to our sum/transfer/broadcast procedure is in fact exactly the number of tensors we should expect according to Maron et al. [2018].

3.3.4 Constructing a Broadcast Tensor for a Specific Partition

Algorithms 1 and 2 detail the steps involved in constructing a permutation equivariant layer between $k_1 \to k_2$'th order permutable tensors. Now we describe the general rule for how entries of the output tensor can be constructed.

If we have $T_1 \in \mathbb{R}^{n^{k_1}}, T_2 \in \mathbb{R}^{n^{k_2}}$, for a given partition partition $\mathcal{P} = \{S_1, \ldots, S_{|\mathcal{P}|}\}$, where the S_i's are non-intersecting subsets of $\{1, 2, \ldots, k_1 + k_2\}$, let $\psi : \{1, 2, \ldots, k_1 + k_2\} \to \{1, 2, \ldots, |\mathcal{P}|\}$ map elements in the base set to the index of the subset containing the element in the partition: $\psi(a) = b$ where $a \in S_b$.

Then, using this function ψ, the broadcast rule for $\mathcal{P}$ determines that the following entries of T_2 have the given contribution from a summation tensor of T_1:

$$[T_2]_{i_{\psi(1)},\ldots,i_{\psi(k_2)}} \leftarrow \lambda \sum_{i_z : z \notin \{\psi(j) | j \in [k_2]\}} [T_1]_{i_{\psi(k_2+1)},\ldots,i_{\psi(k_1+k_2)}}$$

for all entries $i_j \in \{1, \ldots n\}$, and $i_1 \neq i_2 \neq \ldots \neq i_{|\mathcal{P}|}$. The summation is applied over indices that are transfer variables, which are precisely the indices that are not in the same subsets of indicated by the output indices (the first k_2 indices). We find it easiest to understand the construction of the broadcast tensors through examples so we provide two more examples to make the output rule slightly more approachable.

Example 14. *let $T_2 \in \mathbb{R}^{n^3}$ be a 3rd order tensor. Suppose we want to construct a $3 \to 3$ equivariant layer. We consider partitions of $\{1, \ldots, 6\}$. Let $\mathcal{P} = \big\{\{1, 4\}, \{2, 3\}, \{5, 6\}\big\}$.*

There is 1 *transfer variables for* $\mathcal{P}$ *corresponding to the subset containing* 1 *and* 4 *that span the* k_1 *and* k_2 *divide. The summation tensor is a* 1*st order permutable tensor:* $\sum_{i_3} T_{i_1,i_3,i_3}$ *and gets broadcast to* T_2 *as follows:*

$$[T_2]_{i_1 i_2 i_2} \leftarrow \lambda \sum_{i_3} [T_1]_{i_1 i_3 i_3}$$

Example 15. *Using the same third order tensors in example (14), consider the partition* $\mathcal{P} = \big\{\{1,2\},\{3,4\},\{5,6\}\big\}$. *Here we have a transfer variable that links the first index of* T_1 *to the last index of* T_2. *The summation tensor is identical to the summation tensor in the previous example, so the only difference is how we broadcast the summation tensor to the output tensor.*

$$[T_2]_{i_1 i_1 i_2} \leftarrow \lambda \sum_{i_3} [T_1]_{i_2,i_3,i_3}.$$

3.3.5 Making the Layer Learnable

Once we have constructed each of the necessary broadcast tensors, the output tensor is just a linear combination of each of these broadcast tensors. Given an input $\mathbb{R}^{n^{k_1}}$ tensor. By constructing each of the $B(k_1 + k_2)$ tensors according to the sum/transfer/broadcast framework and stacking the results we end up with a multidimensional array of shape $n^{k_2} \times 1 \times B(k_1 + k_2)$, which can be mapped back to a tensor of shape $n^{k_2} \times 1$ by taking a linear combination of the stack of $B(k_1 + k_2)$ tensors.

If the input dimensions were instead (n^{k_1}, d_1) we would have an intermediate tensor of shape $(n^{k_2}, d_1 B(k_1 + k_2))$ which can subsequently be mixed to (n^{k_2}, d_2), where d_2 is the desired output dimension size. Algorithm 2 provides the pseudocode for a $k_1 \to k_2$ permutation equivariant layer. The easiest way to perform this linear mixing on line 3 of Algorithm (2) is with an Einstein summation. For example, if $\widehat{X} \in \mathbb{R}^{n^3 \times d_1 \times k}$ and $M \in \mathbb{R}^{d_1 \times k \times d_2}$ is the learnable weight matrix, then we can linearly mix the last two indices

of $\widehat{X}$ with the following Einstein summation: `torch.einsum("ijkde,def->ijkf", X, M)`.

Algorithm (2) says to compute the broadcast tensor associated with each partition, but this is not strictly necessary. We can pick a subset of the possible partitions to use in our equivariant layer if we have some idea of which sort of interactions are important and which are negligible.

Algorithm 1: `construct_ops` for a $k_1 \to k_2$'th order layer

Input : $k_1 \in \mathbb{N}$, the order of the input tensor, $k_2 \in \mathbb{N}$, the order of the output tensors $\mathbf{X} \in \mathbb{R}^{n^{k_1} \times d_1}$, a k_1th order permutable tensor

Output: List of length $B(k_1 + k_2)$ of permutable tensors in $\mathbb{R}^{n^{k_2} \times d_1}$

```
lst = [ ]
foreach Partition P ∈ Partitions of (k1 + k2) do
    Construct X^P, the appropriate broadcast tensor of X corresponding to P
     according to section 3.3.4
    lst.append(X^P)
end foreach
return lst
```

Algorithm 2: $k_1 \to k_2$ Permutation Equivariant Layer

Input : $\mathbf{X} \in \mathbb{R}^{n^{k_1} \times d_1}$, a k_1th order permutable tensor
$\mathbf{M} \in \mathbb{R}^{d_1 \times B(k_1+k_2) \times d_2}$, a learnable weight matrix

Output: $\mathbf{Z} \in \mathbb{R}^{n^{k_2} \times d_2}$, a k_2th order permutable tensor

1 Construct list of $B(k_1 + k_2)$ intermediate tensors : `ops` $\leftarrow$ `construct_ops`$(k_1, k_2, \mathbf{X})$

2 Stack layers to produce $\widehat{\mathbf{X}} \in \mathbb{R}^{n^{k_2} \times d_1 \times B(k_1+k_2)}$: $\widehat{\mathbf{X}} \leftarrow$ `stack(ops)`

3 Linearly mix $\widehat{\mathbf{X}}$ with $\mathbf{M}$ along $\widehat{\mathbf{X}}$'s last two indices: $\mathbf{Z} \leftarrow \widehat{\mathbf{X}} \cdot \mathbf{M}$

4 **return** $\mathbf{Z}$

3.4 EQUIVARIANT LAYER DETAILS

3.4.1 Complexity of Operations

There are two parts to computing a k_1'th to k_2'th order equivariant layer: computing the intermediate broadcast tensors and computing the linear mixing of these layers. In general,

for a k_1th to k_2th order layer, we first construct intermediate summation tensors of order $k_1 - n_s$ where $n_s \leq k_1$ is the number of summation indices. The complexity of this constructing this summation tensor involves summing over the n_s summation indices which is a $O(n^{n_s})$ operation. Recall from the main body of our paper that if we have n_s summation indices, there are $\binom{k_1}{n_s} B(n_s)$ total summation tensors. So the cost of computing all required summation tensors is: $\sum_{n_s=0}^{k_1} \binom{k_1}{n_s} B(n_s) O(n^{n_s})$.

The complexity of the constructing the broadcast tensor from a given summation tensor is not quite as obvious. Most of the broadcast tensors are constructed by applying `torch.expand` on the relevant summation tensor to broadcast the summation tensor across various dimensions of the output tensor. `torch.expand` (in contrast to `torch.repeat` or `torch.tile`) does not allocate new memory–it only changes the view of a tensor, which is a constant time operation. The computation cost for doing the appropriate broadcasting operations is dominated by the cost of constructing the summation tensors, and the cost of the linear mixing of the broadcast tensors.

In general, the memory overhead for a linear permutation equivariant layer between $k \to k$th order tensors is $O(n^k) \times B(2k)$. We can think of this as being a constant times the k'th order complexity that arises from dealing with a k'th order tensor.

3.4.2 Linearly Mixing the Broadcast Tensors

Suppose our input for a $k_1 \to k_2$ permutation equivariant layer is a tensor living in $\mathbb{R}^{b \times n^{k_1} \times d_{in}}$, where b is the batch size. After constructing all possible broadcast tensors, we have a tensor of shape $b \times n^{k_2} \times d_{in} \times B(k_1 + k_2)$. This can now be mixed with a linear layer that mixes the $d \times B(k_1 + k_2)$ features amongst themselves. Let the output dimension be d_{out}. We apply an Einstein summation to perform the linear mixing. For instance:

- For a $2 \to 2$ permutation equivariant layer, the input X of the layer has shape $b \times n \times n \times d_{in}$. After constructing and stacking the $B(2+2) = 15$ broadcast tensors, we

have a tensor of shape $\widehat{X} \in \mathbb{R}^{b \times n \times n \times d_{in} \times 15}$. Let $M \in \mathbb{R}^{d_{in} \times 15 \times d_{out}}$ be the coefficient matrix that linearly mixes the broadcast tensors. Our output 2nd order tensor would be the result of the following Einstein summation:

$X_{out} =$ `einsum("bijde,def->bijf",` $\widehat{X}, M$`)`

- For a $3 \to 3$ permutation equivariant, the input X has shape $b \times n \times n \times n \times d_{in}$. The stack of broadcast tensors will be $\widehat{X} \in \mathbb{R}^{b \times n \times n \times n \times d_{in} \times 203}$ ($B(3+3) = 203$). The coefficient matrix is $M \in \mathbb{R}^{d_{in} \times 203 \times d_{out}}$. Similar to the $2 \to 2$ case, our output 3nd order tensor would be the result of the following Einstein summation:

  $X_{out} =$ `einsum("bijkde,def->bijkf",` $\widehat{X}, M$`)`

Theorem 6. *Linear combinations of k'th order permutable tensors of size* $\mathbb{R}^{n^k \times 1}$ *are k'th order permutable tensors.*

Proof. Taking the scalar multiple of a permutable tensor is an elementwise operation. Taking the sum of two permutable tensors is also an elementwise operation. Elementwise operations commute with the permutation action so we are done. □

This theorem tells us we can freely take linear combinations of broadcast tensors without worrying about breaking permutation equivariance. All of our writing on k'th order permutable tensors of size $\mathbb{R}^{n^k \times 1}$ extends to permutable tensors of arbitrary feature dimension size $\mathbb{R}^{n^k \times d}$.

3.5 Implementation

Figure 3.1 and 3.2 show the PyTorch code for constructing the $2 \to 2$ and $3 \to 3$ broadcast tensors. These broadcast tensors are then used in the $2 \to 2$ and $3 \to 3$ equivariant layers respectively (Figure 3.3 and 3.4). Once the equivariant layers are implemented as `nn.Module` classes, they can be slotted into neural networks and we no longer need to worry about their implementation.

3.5.1 Architecture

The equivariant layers we propose take in as input a k_1'th order permutable tensor and return a k_2'th order permutable tensor. We will usually have $k_1 = k_2$. Another thing to note is that generally our data just comes in the form of first order permutable tensors; it is actually quite rare to have pairwise, or triplet-wise features explicitly. If we are interested in taking k-ary interactions into account in our model, how do we first construct a k'th order permutable tensor from our data? It turns out that one natural option is to take k-ary products of our data. This is conceptually similar to the tensor product and elementwise product operations proposed by Kondor et al. [2018b], which retains the coveted permutation equivariance property. For $X \in \mathbb{R}^{n \times d}$, we can construct a kth order permutable tensor denoted $X^{(k)}$ with the following entries:

$$X^{(k)}_{i_1,i_2,\ldots,i_k,p} = \prod_{l=1}^{k} X_{i_l,p}.$$

for $i_1, \ldots, i_k \in \{1, 2, \ldots, n\}$ and $p \in \{1, \ldots, d\}$. In practice, this outer product looking operation is something we can implement in PyTorch or TensorFlow using Einstein summations.

For our second order and third order architectures, we followed the following general architecture:

1. Encode each item x_i of the set with a shared neural network ϕ (ex: a ResNet for images, row-wise MLP, row-wise linear layer)

2. Construct a 2nd (or 3rd) order tensor through a 2nd(or 3rd) order product:

   ```
   second_order = torch.einsum("bid,bjd->bijd", x, x)
   ```

   ```
   third_order = torch.einsum("bid,bjd,bkd->bijkd", x, x, x)
   ```

3. Apply $2 \to 2$ (or $3 \to 3$) permutation equivariant layer, followed by a ReLU. This can be repeated multiple times.

4. Apply a permutation invariant pooling operation (sum or mean) over the entity indices to get an embedding vector of the entire set

5. Decode the set embedding vector with a linear layer or multilayer perceptron to produce the final output

Figures (3.1) and (3.2) demonstrate how to construct the various summation tensors that can be linearly mixed for 2nd and 3rd order tensors. Figures (3.3) and (3.4) provide the implementation of the $2 \to 2$ and $3 \to 3$ equivariant layers. The PyTorch pseudocode in Figures (3.5) and (3.6) provide examples of the general structure of 2nd and 3rd order permutation equivariant networks. We refer to a k'th order permutation equivariant network as permutation invariant network that uses a $k \to k$ equivariant layer. In the pseudocode, `Eq2to2` denotes a $2 \to 2$ permutation equivariant layer, and likewise `Eq3to3` denotes a $3 \to 3$ permutation equivariant layer. The pseudocode uses fully connected layers for the encoder and decoder for brevity but in practice, we often use 2 layer multilayer perceptrons.

```
def ops_2_to_2(inputs, normalize=False):
    '''
    Construct the 15 broadcast tensors for a 2 -> 2 equivariant layer
    '''
    N, D, m, m = inputs.shape

    # Summation tensors
    diag_part = torch.diagonal(inputs, dim1=-2, dim2=-1) # N x D x m
    sum_diag_part = diag_part.sum(dim=2, keepdims=True) # N x D x 1
    sum_rows = inputs.sum(dim=3) # N x D x m
    sum_cols = inputs.sum(dim=2) # N x D x m
    sum_all = inputs.sum(dim=(2,3)) # N x D

    # Broadcast the summation tensors
    ops = [None] * (15 + 1)
    ops[1]  = torch.diag_embed(diag_part) # N x D x m x m
    ops[2]  = torch.diag_embed(sum_diag_part.expand(-1, -1, dim))
    ops[3]  = torch.diag_embed(sum_rows)
    ops[4]  = torch.diag_embed(sum_cols)
    ops[5]  = torch.diag_embed(sum_all.unsqueeze(-1).expand(-1, -1, dim))
    ops[6]  = sum_cols.unsqueeze(3).expand(-1, -1, -1, dim)
    ops[7]  = sum_rows.unsqueeze(3).expand(-1, -1, -1, dim)
    ops[8]  = sum_cols.unsqueeze(2).expand(-1, -1, dim, -1)
    ops[9]  = sum_rows.unsqueeze(2).expand(-1, -1, dim, -1)

    ops[10] = inputs
    ops[11] = torch.transpose(inputs, 2, 3)
    ops[12] = diag_part.unsqueeze(3).expand(-1, -1, -1, dim)
    ops[13] = diag_part.unsqueeze(2).expand(-1, -1, dim, -1)
    ops[14] = sum_diag_part.unsqueeze(3).expand(-1, -1, dim, dim)
    ops[15] = sum_all.unsqueeze(-1).unsqueeze(-1).expand(-1, -1, dim, dim)

    return torch.stack(ops[1:], dim=2)
```

Figure 3.1: Constructing the broadcast tensors for a $2 \to 2$ equivariant layer

```
def ops_3_to_3(inputs):
    '''
    Construct a minimal subset (20) of the 3 -> 3 broadcast tensors
    '''
    N, D, m, m, m = inputs.shape

    # Summation tensors
    sum_all = inputs.sum(dim=(-1, -2, -3))
    sum_c1 = inputs.sum(dim=-1)
    sum_c2 = inputs.sum(dim=-2)
    sum_c3 = inputs.sum(dim=-3)
    sum_c12 = inputs.sum(dim=(-1, -2))
    sum_c13 = inputs.sum(dim=(-1, -3))
    sum_c23 = inputs.sum(dim=(-2, -3))

    # Broadcast the summation tensors
    ops = [None] * 20
    ops[1] = sum_all.view(N, D, 1, 1, 1)
                    .expand(-1, -1, dim, dim, dim) / (m * m * m)

    ops[2]  = sum_c1.unsqueeze(-1).expand(-1, -1, -1, -1, m) / m
    ops[3]  = sum_c1.unsqueeze(-2).expand(-1, -1, -1, m, -1) / m
    ops[4]  = sum_c1.unsqueeze(-3).expand(-1, -1, m, -1, -1) / m
    ops[5]  = sum_c2.unsqueeze(-1).expand(-1, -1, -1, -1, m) / m
    ops[6]  = sum_c2.unsqueeze(-2).expand(-1, -1, -1, m, -1) / m
    ops[7]  = sum_c2.unsqueeze(-3).expand(-1, -1, m, -1, -1) / m
    ops[8]  = sum_c3.unsqueeze(-1).expand(-1, -1, -1, -1, m) / m
    ops[9]  = sum_c3.unsqueeze(-2).expand(-1, -1, -1, m, -1) / m
    ops[10] = sum_c3.unsqueeze(-3).expand(-1, -1, m, -1, -1) / m
    ops[11] = sum_c12.view(N, D, m, 1, 1).expand(-1, -1, -1, m, m) / (m*m)
    ops[12] = sum_c12.view(N, D, 1, m, 1).expand(-1, -1, m, -1, m) / (m*m)
    ops[13] = sum_c12.view(N, D, 1, 1, m).expand(-1, -1, m, m, -1) / (m*m)
    ops[14] = sum_c13.view(N, D, m, 1, 1).expand(-1, -1, -1, m, m) / (m*m)
    ops[15] = sum_c13.view(N, D, 1, m, 1).expand(-1, -1, m, -1, m) / (m*m)
    ops[16] = sum_c13.view(N, D, 1, 1, m).expand(-1, -1, m, m, -1) / (m*m)
    ops[17] = sum_c23.view(N, D, m, 1, 1).expand(-1, -1, -1, m, m) / (m*m)
    ops[18] = sum_c23.view(N, D, 1, m, 1).expand(-1, -1, m, -1, m) / (m*m)
    ops[19] = sum_c23.view(N, D, 1, 1, m).expand(-1, -1, m, m, -1) / (m*m)
    return torch.stack(ops[1:], dim=2)
```

Figure 3.2: Constructing a subset of the broadcast tensors for a $3 \to 3$ equivariant layer

```
class Eq2to2(nn.Module):
    def __init__(self, in_dim, out_dim):
        super(Eq2to2, self).__init__()
        self.basis = 15 # Bell(2+2) = 15
        self.out_dim = out_dim
        self.in_dim = in_dim
        self.coefs = nn.Parameter(torch.zeros(in_dim, out_dim, self.basis))
        self.bias = nn.Parameter(torch.zeros(1, out_dim, 1, 1))

    def forward(self, inputs):
        ops = ops_2_to_2(inputs)
        output = torch.einsum('dsb,ndbij->nsij', self.coefs, ops)
        output = output + self.bias
        return output
```

Figure 3.3: Implementation of $2 \to 2$ equivariant layer

```
class Eq3to3(nn.Module):
    def __init__(self, in_dim, out_dim):
        super(Eq3to3, self).__init__()
        self.basis = 19  # use minimal 19 partitions
        self.in_dim = in_dim
        self.out_dim = out_dim
        self.coefs = nn.Parameter(torch.zeros(in_dim, out_dim, self.basis))
        self.bias = nn.Parameter(torch.zeros(1, out_dim, 1, 1, 1))

    def forward(self, x):
        ops = ops_3_to_3(x)
        # in/out/basis, batch/in/basis/ijk
        output = torch.einsum('dsb,ndbijk->nsijk', self.coefs, ops)
        output = output + self.bias
        return output
```

Figure 3.4: Implementation of $3 \to 3$ equivariant layer

```
from perm_eq_layers import Eq2to2, Eq3to3
class Eq2Net(nn.Module):
    def __init__(self, nin, nhid, nout):
        self.enc = nn.Linear(nin, nhid)
        self.eq2a = Eq2to2(nhid, nhid)
        self.eq2b = Eq2to2(nhid, nhid)
        self.dec = nn.Linear(nhid, nout)

    def forward(self, x):           # input shape: B x N x d
        x = self.enc(x))                          # B x N x h
        x = torch.einsum('bid,bjd->bijd', x, x)  # B x N x N x h
        x = F.relu(self.eq2a(x))                  # B x N x N x h
        x = F.relu(self.eq2b(x)).                 # B x N x N x h
        x = x.sum(axis=(1, 2))                    # B x h
        x = self.dec(x)                           # B x o
        return x
```

Figure 3.5: Implementation of a 2nd order permutation equivariant network with two $2 \to 2$ equivariant layers. The encoder and decoders can also be multilayer perceptrons.

```
class Eq3Net(nn.Module):
    def __init__(self, nin, nhid, nout):
        self.enc = nn.Linear(nin, nhid)
        self.eq3a = Eq3to3(nhid, nhid)
        self.eq3b = Eq3to3(nhid, nhid)
        self.dec = nn.Linear(nhid, nout)

    def forward(self, x):                 # input shape: B x N x d
        x = self.enc(x)                                  # B x N x h
        x = torch.einsum('bid,bjd,bkd->bijkd',x,x, x)  # B x N x N x N x h
        x = F.relu(self.eq3a(x))                         # B x N x N x N x h
        x = F.relu(self.eq3b(x)).                        # B x N x N x N x h
        x = x.sum(axis=(1, 2, 3))                        # B x h
        x = self.dec(x)                                  # B x o
        return x
```

Figure 3.6: Implementation of a 3rd order permutation equivariant network with two $3 \to 3$ equivariant layers.

CHAPTER 4

EXPERIMENTS

We evaluated our framework for permutation equivariance by considering equivariant architectures on four set learning tasks: classifying poker hands, counting unique elements from a set of images, and predicting the efficacy of a set of drugs in inhibiting E. coli growth and jet tagging.

4.1 Predicting Poker Hands

A standard playing card deck has 52 cards. Each card has a suit (diamonds, clovers, hearts, or spades) and a numeric value $\{2, 3, \ldots, 10, J, Q, K, A\}$. There are ten official poker hands: high card, pair, three of a kind, two-pair, straight, flush, full-house, straight flush, four of a kind, and royal flush. Given a five card hand of playing cards, our task is to classify the hand as one of the ten possible poker hands. This is a set classification task, since every poker hand's classification does not depend on the way the cards are ordered within a hand. Naturally, we would like to use a permutation equivariant networks to perform the classification.

We use the Poker hand dataset from the UCI repository [Dua and Graff, 2017] which contains a $50,000$ five card hands for the train set, and $950,000$ hands for the test set. Given the success of first order permutation equivariant architectures such as PointNet and Deep Sets, it is reasonable to ask if higher order permutation equivariant layers are even necessary. To answer this question, we evaluate first, second and third order equivariant networks on this poker classification task.

The three models we tested have the following architectures:

- 1st order network (Deep Sets):
  - encode each input card with a 2 layer multilayer perceptron

  - apply two successive Deep Sets permutation equivariant layers
  - Sum pool over the sets to create a set representation vector
  - Apply a final 2 layer multilayer perceptron to produce the class prediction

- 2nd order network (See `Eq2Net` in Figure 3.5)
  - encode each input card with a 2 layer multilayer perceptron as in the 1st order network
  - compute a 2nd order outer product
  - apply two $2 \to 2$ equivariant layers with ReLU activations
  - sum the 2nd order tensor over the two entity dimensions
  - apply a final 2 layer multilayer perceptron to produce the class prediction

- 3rd order network (See `Eq3Net` in Figure 3.6)
  - encode each input card with a 2 layer multilayer perceptron as in the 1st order network
  - compute a 3rd order outer product
  - apply two successive $3 \to 3$ equivariant layers with ReLU activations
  - sum the 3rd order tensor over the three entity dimensions
  - apply a final 2 layer multilayer perceptron to produce the class prediction

As we can see from the results in Table 4.1, second and third order networks effectively classify all hands correctly, use fewer parameters and learn an effective model in fewer epochs than the first order model. The 3rd order network takes longer to train, but that is reasonable given the cubic runtime costs of computing third order outer products and subsequent operations on a 3rd order tensor. With longer training and a larger network (more layers, wider layers, etc), we would likely see the Deep Sets/1st order architecture achieve the same accuracy as the higher order networks.

Table 4.1: Model results on poker hand prediction

Model	Accuracy	Parameters	Time per epoch	Epochs
1st Order Network	0.920($\pm$0.003)	298k	19s	200
2nd Order Network	0.999($\pm$0.001)	20k	11s	100
3rd Order Network	0.999($\pm$0.001)	11k	96s	100

Table 4.2: Poker Model Hyperparameters

Parameter	1st Order	2nd Order	3rd Order
Learning Rate	0.001	0.001	0.001
Batch Size	256	256	256
# of Equivariant Layers	2	2	2
Hidden Dimension	256	24	16

4.2 Counting Unique Elements of a Set

For our second set of experiments, we use the Omniglot [Lake et al., 2015] dataset which contains 1623 different handwritten characters from 50 different languages to predict the number of different characters are in a set of images. This task was first introduced by Lee et al. [2019a].

For each batch in our training data we sample a set size n uniformly from $\{6, 7, \ldots, 10\}$. In the batch, we sample a unique character count c in $\{1, \ldots, n\}$ and then sample n images that have exactly c unique characters among them from the training portion of the Omniglot dataset. We compare a Deep Sets architecture [Zaheer et al., 2017], and Set Transformer architecture [Lee et al., 2019a] against our 2nd order (see `Eq2Net`/`Eq3Net`) and 3rd order permutation equivariant networks. Following the experimental setup described by Lee et al. [2019a] and Shi et al. [2020], we train each of the networks to perform Poisson regression to predict the number of unique characters and minimized the negative log likelihood.

In each network, we use a basic convolutional neural network to encode the images,

before feeding the resulting set of embedded images into permutation equivariant layers. We used the same basic CNN encoder: four convolution layers with ReLU nonlinearities in between each convolution, reshape the final convolution output to a vector and apply a fully connected layer to produce a d dimensional representation for each image of the set. Each of the convolution layers has a kernel size of 3, stride length of 3 and use 8 filters. Using additional channels, additional layers, and/or batch normalization in the CNN embedding module did not affect the prediction accuracy so we kept the CNN embedding module's architecture the same for all experiments.

For the second and third order equivariant networks, we use the aforementioned CNN to embed the images. The set of image representation vectors are then used in a 2nd/3rd order outer product to produce the requisite 2nd/3rd order permutable tensor for the 2nd/3rd order equivariant layer.

Table 4.3: Test accuracy on unique characters task

Model	Accuracy	Parameters	Time per minibatch
Deep Sets	0.62($\pm$0.011)	298k	0.121s
Set Transformer	0.74($\pm$0.021)	216k	0.128s
2nd Order Network (ours)	0.72($\pm$0.018)	139k	0.128s
3rd Order Network (ours)	0.69($\pm$0.014)	96k	0.129s

Table 4.4: Hyperparameters for unique characters experiments

Model	Batch Size	Dropout	Embedding dim	Hidden dim
Deep Sets	32	0	256	256
Set Transformer	32	0	128	128
2nd Order Network (ours)	32	0	64	128
3nd Order Network (ours)	32	0	64	128

4.3 Predicting the Efficacy of Drug Cocktails

Predicting the efficacy of drug combinations is a common problem in pharmacology. Researchers consider the problem of finding the optimal subset of drugs to inhibit the growth of E.coli and tuberculosis in Tekin et al. [2018], Katzir et al. [2019], Cokol et al. [2017]. The goal in these experiments, broadly, is to predict the efficacy of unseen drug combinations. This is a challenging problem due to to the sheer number of combinations of drugs and dosage levels possible.

We use data collected by Tekin et al. [2018], which consists of measurements of E.coli growth after it had been treated with up to five antibiotic drugs out of a predetermined set of eight antibiotics. For every set of antibiotics tested, each of the antibiotics in the set was tested at three dosage levels. The dataset contains three measurements of the bacteria growth as proportion of the control experiment's growth for every combination of up to five drugs, and each possible dosage level for each drug. There are measurements for 24 single, 306 pair, 2403 triplet, 9639 quadruplet and 13608 quintuplet drug combinations.

We use the median of the three measurements as the target value to predict for each drug combination and minimize the mean squared error. Our training set consisted of all the experiments for up to 4 drugs and 80% of the size five drug combinations. The test set is the remaining 20% of size five drug combinations. We construct neural networks that uses $2 \to 2$ and $3 \to 3$ permutation equivariant layers and compare it against Deep Sets and Set Transformer based architectures. We train for a maximum of $12,000$ epochs for all experiments.

Table (4.5) shows the average MAE and RMSE over five random seeds with the standard deviations. The 2nd and 3rd order networks are comparable to the set transformer while using a fraction of the parameters of both baseline models. Third order networks take the longest of the four models, which is not surprising since we have to construct third order tensors to even use the third order layers.

Table 4.5: Test MAE, RMSE on drug combination efficacy prediction task

Model	MAE	RMSE	Parameters	Time per epoch
Deep Sets (sum)	6.00($\pm$0.04)	8.31($\pm$0.11)	1,068,833	0.61s
Set Transformer	5.78($\pm$0.12)	8.21($\pm$0.20)	750,881	0.73s
2nd Order Network (ours)	5.80($\pm$0.09)	8.14($\pm$0.14)	259,617	0.70s
3rd Order Network (ours)	5.84($\pm$0.06)	8.08($\pm$0.08)	227,105	1.50s

4.4 Jet Tagging

In high energy physics, a jet is a spray of scattered particles resulting from a collision events that occur in particle colliders. Jets can be initiated by different elementary particles (ex: quarks, gluons, hadrons, etc) and the resulting jets properties will differ based on the originating particle.

The top quark tagging dataset [Kasieczka et al., 2019] contains a training set of 1.2 million events (jets), 400k validation events, and 400k test events. Each jet consists of a set of around 100 particles and each particle has an associated 4-dimensional feature vector containing the following information:

- logarithm of the particle's energy
- logarithm of the particle's p_T
- difference in pseudorapidity between the particle and the jet axis
- difference in azimuthal angle between the particle and the jet axis.

The learning task is to classify each jet as either a quark or gluon jet. This is effectively a binary set-classification task. Thus, we need to construct a permutation equivariant neural network $f : \mathbb{R}^{n\times 4} \to \{0, 1\}$, where n is the number of particles in the jet.

In this dataset, our sets have size $O(100)$. For $n = 100$ elements in a set, it is prohibitively expensive to construct 3rd order equivariant networks as even computing the 3rd order tensor

product ends up being too computationally expensive. However, we do manage to try the 2nd order architectures on this learning task. The previous three experiments gave some indication about the effectiveness of second and third order networks. In this experiment, we hope to see more compelling evidence for the use of $2 \to 2$ equivariant layers. To this end, we tested four models:

- a baseline Deep Sets architecture with a hidden dimension of 256
- a second order permutation equivariant network with hidden/intermediate dimensions of size 32. This architecture is effectively the same as the architecture displayed in Figure 3.5) except we only use one $2 \to 2$ equivariant layer.
- 2D Tensor Network with a hidden dimension of 32: This network identical to the second order permutation network above except we omit the $2 \to 2$ layer here
- 2D Tensor Network with a hidden dimension of 128

The Deep Sets baseline is a useful one to have, especially since first order methods generally train faster, even if the hidden dimensions of the constituent parts of the model (the hidden dimension of the fully connected layers in the multilayer perceptron encoder or decoder) were wider than the hidden dimensions used in the 2nd order networks. We also apply a so-called "2D Tensor Network" on the task. The idea for this baseline model is see whether the $2 \to 2$ equivariant layer offers any utility or if the 2nd order outer product is the more important contributor to improving the model.

Existing approaches to the top tagging problem in the literature treat the data as a set of items and apply permutation invariant architectures including Energy Flow Networks (a Deep Sets inspired architecture)[Komiske et al., 2019], ParticleNet (a dynamic graph neural network architecture) [Qu and Gouskos, 2020], and even transformers based architectures [Mikuni and Canelli, 2020] on the problem. Table 4.6 shows our final results on the task.

Our results show that 2nd order equivariant networks are competitive with existing models from the literature and use much fewer parameters. The $2D$ tensor networks also perform similarly to the existing models in the literature but take more epochs than the 2nd order networks to train. The increased training epochs is offset by the time required for each epoch.

Table 4.6: Test accuracy on jets tagging dataset

Model	Accuracy	AUC	Parameters	Total Epochs	Time per epoch
ParticleNet [Qu and Gouskos, 2020]	0.940	0.985	498k	-	-
ResNeXt Xie et al. [2017]	0.936	0.985	1.4m	-	-
P-CNN [CMS, 2017]	0.930	0.980	384k	-	-
Energy Flow Networks [Komiske et al., 2019]	0.927	0.979	82k	-	-
Lorentz Group Network [Bogatskiy et al., 2020]	0.929	0.964	4.5k	-	-
Energy Flow Polynomials [Komiske et al., 2018]	0.932	0.980	**1k**	-	-
Deep Sets	0.930($\pm$0.010)	0.979	264k	200	1.3min
2nd Order Network (32 hidden)	0.935($\pm$0.012)	0.983	15k	50	20.30mins
2D Tensor Network (32 hidden)	0.931($\pm$0.012)	0.981	5k	125	3.85mins
2D Tensor Network (128 hidden)	0.932($\pm$0.011)	0.981	83k	100	16.5mins

We find that our permutation equivariant architectures perform reasonably well on a variety of baseline tasks. While we cannot definitively say that our models work better or worse in terms of final test accuracy, our models do require much fewer parameters to achieve similar results. Lastly, many of the other models (particularly Bogatskiy et al. [2020], Komiske et al. [2018]) use some physics domain knowledge to better inform their structure of the models. We do not use any domain knowledge or exploit anything about the data beyond its permutation symmetry and are still able to achieve competitive results on this problem with a relatively simple architecture.

4.5 Discussion

From our experiments we have generated a set of useful advice for constructing the networks

- If Deep Sets performs reasonably on the learning problem, it is worth trying a 2nd order

equivariant network as well and seeing if the 2nd order layers provide an additional boost in performance.

- Initializing the parameters of the equivariant layers is challenging. We drew the entries of our equivariant layers from a normal distribution with mean 0 and standard deviation: $\sqrt{\frac{2}{\text{input dim+basis dim+output dim}}}$. However, it is not clear to us whether this is an ideal initialization.

- Dropout was only necessary in the drug efficacy experiment. We believe regularization was necessary in that experiment and not for the other experiments because the drug cocktail data was quite noisy. Without dropout, we overfitted the training data with all the permutation equivariant models.

- We do not actually need to use all the broadcast tensors in the equivariant layers. This is most apparent for the third order networks, where we only constructed 19 out of the $Bell(3+3) = 203$ possible broadcast tensors for a $3 \to 3$ equivariant layer. In the top tagging experiment, we also experiment with using just 10 of the 15 total broadcast tensors in the $2 \to 2$ equivariant layer. In general, as we briefly mentioned earlier, some of the broadcast tensors can be outright omitted if we have some potential domain knowledge about the data we are working with. For instance, if we know that pairwise relationships are symmetric, then in the 2nd order tensor $X \in \mathbb{R}^{n \times n \times d}$, we would expect $X_{ij} = X_{ji}$ and thus the row sums should be exactly equal to the column sums. So we can omit some of these broadcast tensors outright to reduce computation costs.

- The 2nd and 3rd order outer products are a simple to manufacture higher order interactions from our data. Even using just a 2nd order outer product and applying row-wise multilayer perceptrons before applying a permutation invariant summation over the entity indices is a very simple and strong permutation equivariant baseline.

CHAPTER 5

CONCLUSION

5.1 Concluding Remarks

We summarize our contributions as follows:

- We introduced a systematic way of constructing the various parts of permutation equivariant layers
- We applied these permutation equivariant layers to a new problem domain, namely set-learning
- We provided a general architecture and design principles for permutation equivariant architectures and demonstrated their utility across a few domains

We have given a prescription for systematically constructing the possible permutation equivariant linear operations between arbitrary k_1'th order to k_2'th order permutable tensors. As shown by our various experiments, we may want to apply these permutation equivariant layers as intermediate layers in a permutation invariant neural network for set-learning problems.

Our experiments also demonstrate a simple but expressive permutation equivariant function. [Zaheer et al., 2017] and Qi et al. [2017] give a standard architecture for first order permutation equivariance. We propose architectures that use higher order information by first computing a higher order tensor with k-fold outer products to first generate the desired k'th order permutable tensor. This k'th order tensor can subsequently be featurized with a sequence of $k \to k$ permutation equivariant layers and elementwise nonlinearities before collapsing the kth order tensor down to a 0'th order tensor.

5.2 Future Work

There remains much work to be explored on the topic of permutation equivariance. Our experiments showed that considering higher order interactions using 2nd and 3rd order equivariant layers is helpful for a variety of learning tasks. Using these layers, however, impose a nonnegligible computational cost of the models despite the improvements in model expressitivity and parameter reduction. Part of these computational concerns are unavoidable by nature of needing operate on k-fold interaction data (kth order permutable tensors). There has been recent interest in addressing the quadratic complexity of attention layers [Ren et al., 2021, Xiong et al., 2021], which suggest that we may be able to use similar techniques to reduce the quadratic and cubic runtime costs of 2nd and 3rd order layers. We also know that real world data often has sparse interactions; not every single k-sized subset of items will have meaningful interactions so it is often reasonable to construct higher order equivariant layers that take some sparsity into account similar to recent efforts in applying sparsity aware equivariant graph networks [Morris et al., 2022].

On the applications side, we are also interested in seeing how these higher order equivariant layers may be applied to graph data, where higher order interactions between vertices and edges are often of interest. Maron et al. [2018] first conceived of 2nd and higher order equivariant layers to be used in the context of graph learning. As we mentioned in an earlier section, the graph convolution layers can be viewed as an instantiation of a Deep Sets layer applied on incoming messages. As proposed by Kondor et al. [2018b], Vignac et al. [2020], we can also consider higher order message passing which would naturally suggest that we may also want to apply these higher order permutation equivariant layers.

Finally, there is a natural connection between attention layers and the higher order permutation equivariant layers that was recently explored by Kim et al. [2021]. Given the success of attention based models in recent years from NLP [Vaswani et al., 2017], to graph learning [Veličković et al., 2018], to computer vision [Dosovitskiy et al., 2020], we should

try to consider how we can borrow some of the fundamental ideas that made attention mechanism so effective for those domains and see how they might make these permutation equivariant layers more effective, easier to train, and interpretable on other tasks outside of just the set-learning tasks we considered.

Part II

Fourier Bases for Solving Permutation Puzzles

In Part I of this thesis, we were interested in learning on domains that exhibited permutational symmetry. In particular, we wanted to construct permutation equivariant functions. For a linear function on permutable tensors $f : \mathbb{R}^{n^{k_1} \times d_1} \to \mathbb{R}^{n^{k_2} \times d_2}$, f had to satisfy:

$$\pi \cdot f(X) = f(\pi \cdot X).$$

Satisfying this permutation equivariant constraint places restrictions on the type of operations that can occur in such a linear function f.

In Part II of this thesis, we again deal with a domain that exhibits certain symmetries, namely permutation puzzles. Here, the permutation puzzle configurations are elements of some underlying group, often a symmetric group or a cyclic wreath product group. In trying to construct good heuristic functions for solving these puzzles, we naturally come to the question of how to represent functions on the underlying group which reduces to learning functions on the group itself: $f : \mathcal{G} \to \mathbb{R}$. Here we are interested in functions that exhibit certain smoothness properties and economical ways of constructing these functions, which naturally brings us to our proposed solution of using a limited set of Fourier basis functions in representing our function f.

CHAPTER 6

INTRODUCTION

Solving combinatorial puzzles has long captivated AI and ML researchers. Many of these puzzles such as the Rubik's Cube have enormous state spaces that cannot be fully enumerated and brute forced, lending themselves to be used as test beds for more "intelligent" search methods. Most methods to solve these puzzles such as the Rubik's Cube (Fig: 7.1), Pyraminx (Fig: 7.2a), and the Sliding Number Tile Puzzle (Fig 9.1) involve heuristic search such as A^*-search and Iterative Deepening A^*-search (IDA) [Korf, 1985]. It is infeasible to deduce the distance of every puzzle state to the goal state, but often it is possible to have some heuristic function that serves as proxy for the true distance function that can still provide useful guiding information in searching for a solution.

The runtime of these heuristic search methods depends on the quality of the chosen heuristic functions used estimate the distance of a given puzzle state to the goal state. The most effective heuristic functions for permutation puzzles like the Rubik's Cube and the Sliding Number Tile Puzzle use pattern databases [Korf and Taylor, 1996, Korf, 1997, Kociemba, n.d., Felner et al., 2004], which effectively store the distance of various partially solved puzzles to the solution state. Korf [1997] famously solved instances of the Rubik's Cube using a pattern database of all possible corner cube configurations and various edge configurations and has since improved upon the result with an improved hybrid heuristic search [Bu and Korf, 2019].

In recent years, machine learning based methods have become popular alternatives to classical planning and search approaches for solving these puzzles. Brunetto and Trunda [2017] treat the problem of solving the Rubik's cube as a supervised learning task by training a feed forward neural network on 10 million cube samples, whose ground truth distances to the goal state were computed by brute force search. Value based reinforcement learning methods to solve the Rubik's Cube and Sliding Number Tile Puzzle [Bischoff et al., 2013]

are also common. Most recently, McAleer et al. [2018] and Agostinelli et al. [2019] developed a method to solve the Rubik's cube and other puzzles by training a neural network value function through reinforcement learning. They then used this value network in conjunction with A^* search and Monte Carlo tree search to solve the Rubik's cube.

In Part II of this thesis, we focus on learning to solve permutation problems in the reinforcement learning setting, specifically using value functions. The crux of our approach to solving permutation puzzles is to exploit the underlying structure of the puzzle by using specialized basis functions for expressing this value function. There has been a wealth of research in reinforcement learning on constructing basis functions for value function approximation [Keller et al., 2006, Menache et al., 2005]. Common choices for a set of basis functions include radial basis functions [Kretchmar and Anderson, 1997] and Fourier series (sinusoids) [Konidaris et al., 2011]. Proto Value Functions (PVF) [Mahadevan and Maggioni, 2007, 2006, Mahadevan, 2005] try to solve Markov decision processes (MDP) by representing the value function in the eigenbasis of the MDP's graph Laplacian. Their method uses ideas from spectral graph theory which tell us that the smoothest functions on a graph correspond to the lowest eigenvalue eigenvectors of the graph Laplacian. Though our work is conceptually similar to Proto Value Functions, computing the eigenvectors of the graph Laplacian of a permutation puzzle would be a prohibitively expensive $O(n^3)$ operation, where n is the size the puzzle. Our approach uses fundamentally spectral ideas as well, but does not suffer from cubic run-time costs.

What is an appropriate basis for learning functions over our permutation puzzles? It turns out that the puzzles we are interested in solving are groups. Representation theory [Serre, 1977a] tells us there is a natural basis, the Fourier Basis, for expressing functions over a given group. Using this Fourier basis, we show that we can learn linear value functions to permutation puzzles such as Pyraminx, and the 2-by-2 Rubik's cube with far fewer samples and orders of magnitude fewer parameters than a neural network parameterization. Our

work is the first to demonstrate the efficacy of the Fourier Basis in reinforcement learning and the first to propose leveraging the representations of wreath product groups to solve the 2-by-2 cube.

CHAPTER 7

PRELIMINARIES

In this chapter, we will review some of the mathematical concepts that will underpin our approach to solving various permutation puzzles. Some of the terminology such as groups were discussed in the mathematical preliminaries in Part I, but to allow our readers to avoid flipping back, we will repeat some of those definitions here as well.

7.1 Groups

Recall that a group $\mathcal{G}$ is a set of objects endowed with a multiplication operation $\cdot : \mathcal{G} \times \mathcal{G} \to \mathcal{G}$ which satisfies the three properties:

1. the group multiplication is associative
2. there exists an identity element of $\mathcal{G}$, which we denote e, such that for every $g \in \mathcal{G}$, $g \cdot e = e \cdot g = g$
3. for every $g \in \mathcal{G}$ there exists an inverse element $g^{-1} \in \mathcal{G}$.

The state space of a permutation puzzle naturally forms a group, since any legal move can be followed by any other legal move, and also any legal move has an "inverse" which just undoes it.

Definition 7.1.1. *Given a subset of group elements: $S \subseteq G$, we denote by $\langle S \rangle$ the subgroup of $\mathcal{G}$* ***generated by*** *S, i.e., the set of all elements of $\mathcal{G}$ that can be expressed as a finite product of the elements of S and their inverses. We say $\mathcal{G}$ is generated by S if $G = \langle S \rangle$.*

Definition 7.1.2. *Given a group $\mathcal{G}$ and a set of generators $S \subseteq \mathcal{G}$, the* ***Cayley Graph*** *$\Gamma(\mathcal{G}, S)$ is a graph whose vertices correspond to elements of $\mathcal{G}$ (in a one-to-one fashion) and for any two vertices v_g and v_h, there is an edge between v_g and v_h if there is some $s \in S$ such that $s \cdot g = h$.*

Given a puzzle's underlying group $\mathcal{G}$ and legal puzzle moves (i.e., generators) S, solving the puzzle can be recast as finding a path on $\Gamma(G, S)$ from an arbitrary group element $g \in \mathcal{G}$ to a solved state d. Without loss of generality, we take the identity element as the solved state.

Example 16. *The set of legal positions of the Rubik's Cube forms a group. The inverse of any face move is the move twisting the same face in the opposite direction. The solved state is the identity element. Every legal cube position is the result of applying a sequence of* 90 *degree clockwise or counter-clockwise face moves (Front, Back, Left, Right, Up, or Down) to the solved state. Any legal cube state g which is produced by the sequence of face twists:* $m_1 m_2 \dots m_k$*, has an inverse:* $m_k^{-1} \dots m_1^{-1}$.

It turns out that in the case of all the permutation puzzles that we are interested in, the puzzle's group is a so-called **wreath product group**.

Definition 7.1.3 (Product Group)**.** *Given groups* $\mathcal{G}_1$ *and* $\mathcal{G}_2$*, the product of the two groups denoted* $\mathcal{G}_1 \times \mathcal{G}_2$ *is also a group. The identity element in* $\mathcal{G}_1 \times \mathcal{G}_2$ *is* (e_1, e_2)*, where* e_1 *is the identity in* $\mathcal{G}_1$ *and* e_2 *is the identity in* $\mathcal{G}_2$*. The group multiplication in* $\mathcal{G}_1 \times \mathcal{G}_2$ *is given by:*

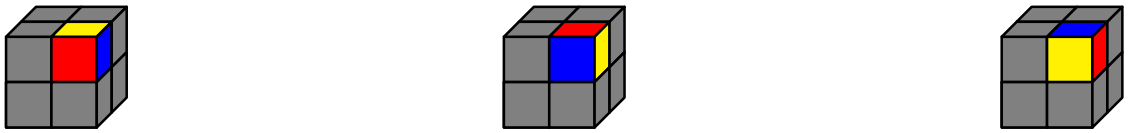
$$(g_1, g_2) \cdot (h_1, h_2) = (g_1 h_1, g_2 h_2)$$

for any $g_1, h_1 \in \mathcal{G}_1$ *and* $g_2, h_2 \in \mathcal{G}_2$.

For any positive integer n, we can take the n-fold product group of $\mathcal{G}$, which we write as: $\mathcal{G}^n = \underbrace{\mathcal{G} \times \ldots \times \mathcal{G}}_{n \text{ times}}$. Elements of $\mathcal{G}^n$ can be written as n-tuples of elements of $\mathcal{G}$.

Definition 7.1.4 (Wreath Product Group)**.** *The wreath product group* $\mathcal{G} \wr \mathbb{S}_n$ *is defined to be the set of elements in* $\mathcal{G}^n \times \mathbb{S}_n$ *with the group multiplication defined as:*

$$(a, \pi) \cdot (b, \sigma) = (a \cdot (\pi \cdot b),\ \pi\sigma),$$



(a) (b) (c)

Figure 7.1: 2-by-2 Cube: the three possible orientations of a given corner cubie.

where the action of $\pi \in \mathbb{S}_n$ *on* $b \in \mathcal{G}^n$ *is defined* $[\pi \cdot b]_i = b_{\pi^{-1}(i)}$.

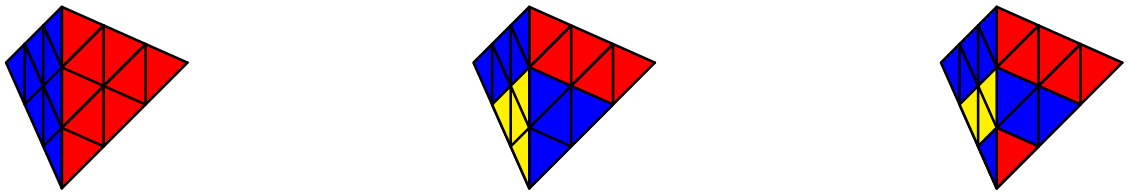
The permutation puzzles that we discuss will be wreath product groups where $\mathcal{G}$ is a cyclic group.

Example 17. *The group of the 2-by-2 Rubik's cube is a subgroup of the wreath product group:* $C_3 \wr \mathbb{S}_8$. *There are eight moveable "cubies" and each cubie has three visible colored facets. Each face twist permutes the eight cubies among themselves while cycling through the three possible orientations of a cubie (as shown in Fig 7.1).*

Example 18. *Pyraminx (Fig: 7.2a) is a three-layered tetrahedron shaped puzzle similar to the Rubik's Cube. The tips and middle layers of the puzzle can be rotated clockwise or counter-clockwise by* $\frac{2\pi}{3}$. *There are 4 tips and 6 edge facets that can be moved. The tips can cycle between three possible orientations and be moved independently of the rest of the layers of the puzzle. Similar to the cubies of the 2-by-2 cube, the six edge facets get permuted amongst themselves and cycle between their two possible orientations (determined by the two colors). The full group is a subgroup of:* $(C_2 \wr \mathbb{S}_6) \times C_3^4$. *In practice, we can ignore the latter* C_3^4 *component of the group because they correspond to the orientations of the four tip pieces, which are trivial to orient correctly.*

7.2 Fourier Analysis on Finite Groups

Classical Fourier analysis gives us the tools to decompose functions on the real line into linear combinations of sinusoidal basis functions. The harmonic analysis of functions on



(a) (b) (c)

Figure 7.2: Suppose the bottom face of the Pyraminx puzzle in Fig 7.2a is yellow. Rotating the puzzle's front two layers clockwise by $\frac{2\pi}{3}$ results in Fig 7.2b. Subsequently, rotating just the front tip counterclockwise by $\frac{2\pi}{3}$ results in Fig 7.2c.

permutation groups is defined using **representations** and similarly gives us the tools to decompose functions on the group into the analogous basis functions. Representations of finite groups have been thoroughly studied in mathematics [Serre, 1977a, Diaconis, 1988], so we will simply give a brief overview of the main concepts from the field that underpin our work.

Definition 7.2.1. *A **representation** ρ of a group $\mathcal{G}$ is a matrix valued function: $\rho : \mathcal{G} \to \mathbb{C}^{d_\rho \times d_\rho}$ that preserves the group structure: $\rho(g \cdot h) = \rho(g)\rho(h)$. d_ρ is referred to as the dimensionality of the representation.*

Example 19. *The permutation matrices of size $n \times n$ comprise a representation for the Symmetric group $\mathbb{S}_n$: $[\rho(\sigma)]_{ij} = \mathbb{1}\{\sigma(j) = i\}$ for $\sigma \in \mathbb{S}_n$ and $1 \leq i, j \leq n$*

We say that two representations ρ_1 and ρ_2 are equivalent if there exists some invertible transformation T such that $\rho_1(g) = T\rho_2(g)T^{-1}$ for all $g \in \mathcal{G}$. We can express this as $\rho_1 \equiv \rho_2$.

Given two representations ρ_1 and ρ_2 of G, we can construct larger representations by taking their direct sum:

$$(\rho_1 \oplus \rho_2)(g) = \left(\begin{array}{c|c} \rho_1(g) & 0 \\ \hline 0 & \rho_2(g) \end{array}\right).$$

We say that a representation ρ of a finite group $\mathcal{G}$ is **reducible** if there exists representations

ρ_1, ρ_2 such that $\rho \equiv \rho_1 \oplus \rho_2$. A representation is **irreducible** if it is not reducible. For a given finite group $\mathcal{G}$, we can construct infinitely many inequivalent reducible representations by taking any number of direct sums of an arbitrary representation ρ. However, $\mathcal{G}$ only has finitely many inequivalent irreducible representations.

Example 20. *The cyclic group C_m has m irreducible representations. They are the complex exponentials: $\chi_k(j) = e^{2\pi ijk/m}$ for $j \in C_m, k \in \{0, 1, ..., m-1\}$ regarded as "1×1 matrices".*

For notational convenience, we will also refer to irreducible representations as "irreps" going forward. The group **Fourier transform** of a function $f : \mathcal{G} \to \mathbb{C}$ at a given irreducible representation ρ is defined as:

$$\widehat{f}_\rho = \sum_{g \in \mathcal{G}} f(g)\rho(g). \tag{7.1}$$

The **inverse Fourier transform** is

$$f(g) = \frac{1}{|\mathcal{G}|} \sum_{\rho \in \mathcal{R}} \mathrm{Tr}[\widehat{f}_\rho^\top \rho(g)]. \tag{7.2}$$

One way of looking at the irreducible representations of a finite group is to view each irreducible ρ is a collection of $d_\rho \times d_\rho$ individual functions $\rho_{ij} : \mathcal{G} \to \mathbb{C}$, with $\rho_{ij}(g) = [\rho(g)]_{ij}$. Recall the property of traces: $\mathrm{Tr}[A^\top B] = \mathrm{vec}(A)^\top \mathrm{vec}(B)$. The inverse Fourier transform of f is just an expansion of f into a linear combination of these ρ_{ij} functions.

Theorem 3. *Serre [1977a] Given a complete set of unitary irreducible representations of a finite group $\mathcal{G}$, the set of matrix entries of these irreducible representations form a complete orthonormal basis for $L(\mathcal{G}) = \{f : \mathcal{G} \to \mathbb{C}\}$, the space of functions on $\mathcal{G}$:*

$$L(\mathcal{G}) = \mathrm{span}\{\sqrt{d_\rho}\, \rho_{ij} \mid 1 \le i, j \le d_\rho\}.$$

Going forward, we will also refer to the irreducible representation basis as the **Fourier**

Table 7.1: Dimensionality of irreps of $C_3 \wr \mathbb{S}_8$

Young Subgroup	Young Subgroup Irrep	Dim
$\mathbb{S}_2 \times \mathbb{S}_3 \times \mathbb{S}_3$		560
$\mathbb{S}_4 \times \mathbb{S}_2 \times \mathbb{S}_2$		420
$\mathbb{S}_2 \times \mathbb{S}_3 \times \mathbb{S}_3$		2240
$\mathbb{S}_3 \times \mathbb{S}_4 \times \mathbb{S}_1$		1680

basis, as is common in existing literature [Huang et al., 2009]. Theorem 1 and the definition of the inverse Fourier transform tell us that we can express any function $f : \mathcal{G} \to \mathbb{C}$ in Fourier space using the matrix entries of the irreducible representations of $\mathcal{G}$ as a basis.

There exist well known constructions for the irreps of $\mathbb{S}_n$ such as Young's Orthogonal Representation (YOR). The irreducible representations of wreath product groups of the form $C_m \wr \mathbb{S}_n$ are constructed by inducing the irreducible representations of Young Subgroups of $\mathbb{S}_n$ up to $\mathbb{S}_n$ [Ceccherini-Silberstein et al., 2014]. In Table 7.1, we show some examples of the irreps of $C_3 \wr \mathbb{S}_8$, the wreath product group associated with the 2-by-2 cube along with their dimensionalities. Chapter 11 describes the process for constructing the irreps of $\mathbb{S}_n$ and the irreps of wreath product groups. For now we just assume that we can evaluate the irreps of the symmetric group and various wreath product groups using some library functions.

CHAPTER 8

REINFORCEMENT LEARNING BACKGROUND

Exact search methods such as breadth first search and depth first search are generally infeasible for puzzles with a large configuration space. We are also interested in the problem of *learning* how to solve these puzzles and how to somehow use some of our domain knowledge of the mathematical structure of these puzzles to solve them. To that end, we do not consider the brute force approach and instead consider the problem in the reinforcement learning setting.

In a classical reinforcement learning task, we have an agent interacting in some environment that produces some signal or reward function to the agent. The broad goal in these tasks is to control the agent in such a way as to maximize its long term reward. As we will see shortly, Markov Decision Processes [Puterman, 1994, Sutton and Barto, 2018] are a simple framework for mathematically defining a reinforcement learning task which naturally suit our goal of learning a how to solve permutation puzzles.

8.1 Markov Decision Processes

A Markov Decision Process is described by a tuple $\mathbb{M} = (S, A, r, T, \gamma)$, where

1. S is the state space. This can be a discrete or continuous space.

2. A is the action space. This can be a discrete or continuous space.

3. r is the reward function. $r : S \times A \to \mathbb{R}$.

4. T is the transition probability function: $T : S \times A \times S \to [0, 1]$. $T(s_1, a_1, s_2)$. denotes the probability of transitioning from s_1 to s_2 after taking action. Some references instead use the notation: $P(s'|s, a) =$ the probability of observing state s' after applying action a to state s.

In deterministic settings where taking a given action at a given state always results in the same state, we have the transition function $\mathcal{T} : S \times A \to S$.

5. $\gamma \in [0, 1]$ is the discount parameter, which determines how much to to down weight future rewards in favor of immediate rewards.

In **deterministic** MDPs, applying action a at state s always results in the same state s so the reward function r can be simplified to $r : S \to \mathbb{R}$.

Definition 8.1.1. *The **return** of a sequence of states $\{s_t\}$, actions $\{a_t\}$, and corresponding rewards $\{r_t\}$ is*

$$G_t = \sum_{k=0}^{\infty} \gamma^k r(s_{t+k}, a_{t+k})$$

In general, we'd like to find a policy $\pi : S \times A \to [0, 1]$ for a Markov Decision process that maximizes the expected return, where this expectation is with respect to the actions chosen by the policy π and the transition dynamics T of the MDP.

We can use a deterministic Markov Decision process to model our permutation puzzles. The state space is the set of all legal puzzle states, the action space is the set of moves. The reward function is

$$r(s_t) = \begin{cases} 1 \text{ if } s \text{ is the solved state} \\ 0 \text{ otherwise} \end{cases}$$

Definition 8.1.2. *The expected return of a state s under a policy π is: given by the value functions $V^\pi : S \to \mathbb{R}$, and $Q^\pi : S \times A \to \mathbb{R}$*

$$V^\pi(s) = \mathbb{E}_\pi[G_t | s_t = s]$$

$$Q^\pi(s) = \mathbb{E}_\pi[G_t | s_t = s, a_t = a]$$

The optimal value function $V^* = \max_\pi V^\pi$ for all $s \in S$, and likewise for Q^* must satisfy the following recurrence, also known as the Bellman Optimality Equation [Bellman, 1957,

Sutton and Barto, 2018]:

$$V^*(s) = \max_{a \in A} \sum_{s'} P(s'|s,a)[r(s,a) + \gamma V^*(s')]$$

$$Q^*(s,a) = \sum_{s'} P(s'|s,a)[r(s,a) + \gamma \max_{a \in A} Q^*(s')]$$

For MDPs with deterministic transitions (i.e.: given a state-action pair (s, a), there is a deterministic subsequent state s') the recurrence for V^* simplifies to:

$$V^*(s) = \max_{a \in A}[r(s,a) + \gamma V^*(\mathcal{T}(s,a))]$$

Given an optimal value function Q then naturally induces a policy, where the action chosen at any given state s is the argmax action a of $Q(s, a)$.

8.2 Value Iteration and Approximate Dynamic Programming

Value iteration [Bellman, 1957] is one approach for learning a value function for solving the MDP is to directly model a solution to the Bellman Optimality equation:

$$V^{\pi}_{i+1}(s) \leftarrow \max_{a \in A} \sum_{s'} r(s,a) + \gamma T(s,a,s') V^{\pi}_i(s')$$

Traditionally, value iteration used a lookup table to represent the $V(s)$ (or $J(s)$) values for all $s \in S$ and iteratively updated each entry of the table until V converged. This method is infeasible when the state space is too large.

Approximate dynamic programming (also referred to as neurodynamic programming) [Bertsekas and Tsitsiklis, 1996] is a framework for computing an approximation to the true value function. The value function can be represented by as neural network, or some other

parameterized model that minimize the difference between the value of a current state/action pair $Q(s, a)$ and its one step look ahead value. This is the approach that underlies the classic Q-learning [Watkins and Dayan, 1992a] update, which at iteration i performs the following update:

$$Q_{i+1}(s_t, a_t) \leftarrow Q_i(s_t, a_t) + (1 - \alpha)(r_t + \max_{a \in A} Q_i(s_{t+1}, a) - Q_i(s_t, a_t)) \tag{8.1}$$

This can be viewed as an online gradient descent update with a step size of $\frac{\alpha}{2}$ on the example (s_t, a_t) for the loss:

$$Loss = (Q_i(s_t, a_t) - (r_t + \gamma \max_{a \in A} Q_i(s_t, a)))^2$$

Deep Q-learning [Mnih et al., 2013, 2015], a now commonly used method in reinforcement learning, uses deep neural networks to approximate the Q value function and essentially iteratively applies the Q-learning gradient step (8.1) in batches of state, action, reward tuples drawn from a cache/replay buffer.

For permutation puzzles, since the state, action transitions are deterministic, we can model our value function V parameterized by some function approximator (e.g. a neural network) with parameters θ and the following loss function that gets minimized by batch stochastic gradient descent over each batch B:

$$Loss(B; \theta) = \frac{1}{|B|} \sum_{s \in B} (V_\theta(s) - (\max_{s' \sim s} r(s') + \gamma V_\theta(s')))^2$$

We focus on using value functions for solving MDPs instead of popular policy gradient based approaches such as actor-critic [Sutton et al., 2000], trust region policy optimization (TRPO) [Schulman et al., 2015], proximal policy optimization [Schulman et al., 2017], due to the relative simplicity (in implementation and interpretability) of a pure value based method. In some sense, value based methods are even more natural for solving permutation

puzzles if we interpret the problem as a graph search since value iteration is effectively a graph diffusion.

CHAPTER 9

RELATED WORK

Permutation puzzles often exhibit symmetries that can be exploited to shrink their state space. In the traditional planning literature, many have proposed various procedures to handle symmetries in heuristic search. Fox and Long [1999, 2002], Pochter et al. [2011] try to detect symmetric states during heuristic search to avoid searching states that are isomorphic to previously seen states. Domshlak et al. [2012] tries to detect symmetries in the transition graph of the puzzle to avoid redundant search as well. While our work deals with permutation puzzles that do have symmetries, we do not explicitly model any of their symmetries since our focus is on demonstrating the efficacy of the Fourier basis for value function approximation.

The Fourier basis of the symmetric group has been used in machine learning and statistics [Diaconis, 1988] for a variety of applications including: learning rankings [Kondor and Barbosa, 2010, Kondor and Dempsey, 2012, Mania et al., 2018], and performing inference over probability distributions on permutations for object tracking [Kondor et al., 2007, Huang et al., 2009, 2008]. These works deal with bandlimited representations of functions over the symmetric group, but do not address reinforcement learning over permutation puzzles.

9.1 Proto-Value Functions

Mahadevan and Maggioni [2007] introduced Proto-Value Functions (PVF) for reinforcement learning. To properly discuss the ideas behind PVF, we must first introduce a few concepts from spectral graph theory [Chung and Graham, 1997]. Given graph $G = (V, E)$ with n vertices, the **graph Laplacian** is the matrix $\mathcal{L} = D - A$, where D is the diagonal matrix with entries $D_i = \sum_{j=1}^{n} A_{ij}$ and A is the (weighted) adjacency matrix of G. $\mathcal{L}$ a symmetric positive semidefinite matrix, so it has a complete orthonormal basis of eigenvectors. Let

$v_1, v_2, \ldots, v_n$ denote the complete orthonormal set of eigenvectors of $\mathcal{L}$ with corresponding eigenvectors in increasing size $0 \leq \lambda_1 \leq \lambda_2 \leq \ldots \leq \lambda_n$.

For any vector $f \in \mathbb{R}^n$, we can view f as a function on the vertices of G where the i'th index of f indicates the function value on the ith vertex. The "smoothness" of f is captured by evaluating its quadratic form with the graph Laplacian:

$$f^\top \mathcal{L} f = \frac{1}{2} \sum_{(i,j) \in E} (f_i - f_j)^2.$$

We say that a graph function is smooth if adjacent nodes have similar function values. The smoothest graph function is the constant function over the graph, which is exactly the first eigenvector: $v_1 = \frac{1}{\sqrt{n}}[1, 1, \ldots, 1]^\top$ with eigenvalue $\lambda_1 = 0$. It is also common in the literature to see smooth functions referred to as low frequency functions. High frequency graph functions then correspond to functions that differ widely across adjacent nodes.

It is natural to consider smooth value functions for reinforcement learning since the value of being in a given state or one of its neighboring state should not vary very much. Mahadevan and Maggioni [2007]'s Proto-Value Functions try to use the low frequency eigenvectors to construct smooth value functions for solving MDPs. Each state of the MDP is a node in the graph G, and two states s_i and s_j have an edge between them if it is possible to traverse from s_i to s_j. Given graph G, they compute the smallest k smoothest eigenvectors (lowest valued eigenvectors) of the graph Laplacian of G. Finally, they estimate a value function for the task using Q-learning [Watkins and Dayan, 1992b] in the basis spanned by the smoothest k eigenvectors.

9.2 Heuristic Search in Fourier Space

Our work builds on Swan [2017]'s which used the irreducible representations of the symmetric group specifically for solving the 8-puzzle (the 3-by-3 sliding number tile puzzle).

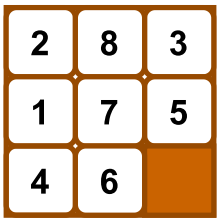


Figure 9.1: The 8-puzzle, also known as the 3-by-3 sliding number tile puzzle. The puzzle starts in a scrambled state and the player tries to arrange the tiles back into sorted order by maneuvering tiles into the empty free slot.

Swan demonstrates that commonly used A^* heuristics for solving the puzzle such as the Hamming Distance [1], and Manhattan distance[2], are bandlimited in the Fourier basis–many of the heuristic functions' Fourier matrices are zero matrices and can be ignored. A heuristic function h for the 3-by-3 sliding number tile puzzle can be written as a function on the symmetric group: $h : \mathbb{S}_n \to \mathbb{R}$. Recall from Chapter 7.2 that h can be expressed in terms of the inverse Fourier transform:

$$\widehat{h}_\rho = \sum_{\sigma \in \mathbb{S}_n} h(\sigma)\rho(\sigma), \qquad \text{for all } \rho \in \mathcal{R}$$
$$h(\sigma) = \frac{1}{n!} \sum_{\rho \in \mathcal{R}} d_\rho \mathrm{Tr}(\widehat{h}_\rho^\top \rho(\sigma))$$

where $\mathcal{R}$ is a complete set of inequivalent irreducible representations of $\mathbb{S}_n$. If h can be expressed in terms of a subset of the irreducible representations, the summation in the inverse Fourier transform (the latter equation) will then span over a reduced set of irreducible representations.

Starting with the non-zero Fourier matrices of the Manhattan and Hamming distance heuristics, Swan uses a derivative-free optimization technique to iteratively update the heuris-

1. The Hamming Distance of a given puzzle state in the sliding number tile puzzle counts the number of tiles that are not in their correct position

2. The Manhattan Distance (or L_1 distance) of the sliding number tile puzzle is the sum of the distances each tile is from its correct position.

tic function determined by these Fourier matrices in order to minimize the number nodes explored during A^*-search.

Our approach differs from Swan's in that we are learning value functions (which can eventually be used with heuristic search) in the reinforcement setting. We also go further than Swan by demonstrating that reinforcement learning using the Fourier basis works on wreath product groups as well the symmetric group.

9.3 Deep Value Networks

Finally, deep neural networks have been used to great success in learning value functions on the Rubik's cube by McAleer et al. [2018] and Agostinelli et al. [2019]. They construct a deep value network that interleaves multiple fully connected layers with residual connections and ReLU nonlinearities. The state of a Rubik's cube is encoded as a 480 dimensional vector that passes through an initial $480 \to 5000$ dimensional fully connected layer, followed by nine fully connected layers of dimension $1000 \to 1000$, and a final $1000 \to 1$ dimensional fully connected output layer. There are residual connections between every other intermediate layer as shown in Figure 9.2 between every two intermediate fully connected layers and ReLU activations after every non-output layer.

Their parameterized value function $V : S \to \mathbb{R}$ is trained to minimize the difference between the value at a given state and the one-step lookahead value in accordance with the Bellman equation (Equation 8.1) over a batch of states B generated by taking a random walk of length N over the cube:

$$Loss(B) = \sum_{s \in B} \left(V(s) - \left(\max_a R(s) + V(\mathcal{T}(s, a)) \right) \right)^2.$$

During testing time, McAleer et al. [2018] and Agostinelli et al. [2019] use the learned value network as the heuristic function in conjunction with A^*-search and Monte Carlo tree search

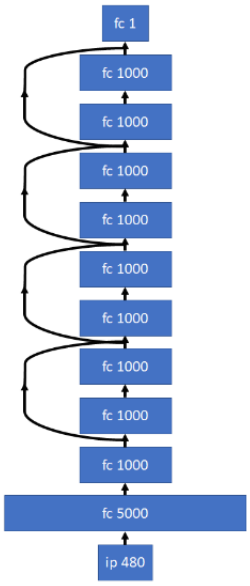


Figure 9.2: Agostinelli et al. [2019]'s deep value network for solving the Rubik's Cube

to solve the cube.

McAleer et al. [2018] and Agostinelli et al. [2019]'s deep value network does perform reasonably well–they are able to solve solve all cubes in their test set given a long enough time budget for their Monte Carlo tree search. One glaring drawback of their approach is that the Rubik's cube is treated as a general reinforcement learning environment–they do not leverage any of the mathematical properties of the puzzle.

CHAPTER 10

LEARNING A VALUE FUNCTION IN THE FOURIER BASIS

For a permutation puzzle whose states are elements of an underlying group $\mathcal{G}$, we propose learning a value function $V : \mathcal{G} \to \mathbb{C}$ in the Fourier basis. Taking inspiration from [Mahadevan and Maggioni, 2007] and Swan [2017], we aim to parameterize the value function as a smooth, bandlimited function using some subset of the irreducible representations of the underlying groups of the permutation puzzles. Ideally we would like to use the Laplacian eigenvectors as a basis for the value function of some of these puzzles, but that is prohibitively expensive for the 2-by-2 cube where the state space is on the order of millions.

Once we pick a subset of irreducible representations of $\mathcal{G}$ to use, learning a value function amounts to learning their respective Fourier matrices. Let $\mathcal{R}$ be a complete set of inequivalent irreducible representations of $\mathcal{G}$ and $I \subseteq \mathcal{R}$ the subset of irreps that we pick. The value function is then approximated as:

$$V(g) = \sum_{\rho \in I} \mathrm{Tr}[\theta_\rho^\top \rho(g)], \tag{10.1}$$

where $\theta_\rho \in \mathbb{C}^{d_\rho \times d_\rho}$ for $\rho \in I$ are the learnable Fourier coefficient matrices. Note that the domain of the parameters may be $\mathbb{C}$ instead of $\mathbb{R}$ since some irreps of groups such as the 2-by-2 cube are defined over $\mathbb{C}$ (from its factor of C_3). Equation (10.1) can also be rewritten as:

$$V(g) = \theta^\top \Big(\bigoplus_{\rho \in I} \mathrm{vec}(\rho(g)) \Big), \tag{10.2}$$

$$\theta = \bigoplus_{\rho \in I} \mathrm{vec}(\theta_\rho) \in \mathbb{C}^{\sum_{\rho \in I} d_\rho \times d_\rho}. \tag{10.3}$$

where vec is the linear operator that converts a matrix into a column vector. The scaling

factors of $\frac{1}{|\mathcal{G}|}$ and d_ρ from the inverse Fourier transform (Eqn. 7.2) have been collapsed into the θ_ρ terms. Using this parameterization of the value function, we can use standard value based reinforcement learning algorithms such as Q-learning or value iteration to learn V. The number of parameters in V is: $\sum_{\rho \in I} d_\rho^2$.

10.1 Low Rank Fourier Matrices

Depending on the dimensionality of the irreducible representations that we use to parameterize V, it might not be feasible to learn dense θ_ρ matrices. We can circumvent this by parameterizing θ_ρ as a low rank matrix: $\theta_\rho = U_\rho W_\rho^\top$, where $U_\rho \in \mathbb{C}^{d_\rho \times k}$, $W_\rho \in \mathbb{C}^{d_\rho \times k}$, and $k \ll d_\rho$. The number of parameters in V is then: $2k \sum_{\rho \in I} d_\rho$.

10.2 Algorithm

Our algorithm for performing value iteration in Fourier space is Algorithm 3. We follow the general structure of deep-Q learning popularized by Mnih et al. [2013, 2015] and Silver et al. [2016]. In each iteration of the main loop, we sample a random walk starting from the goal state of our permutation puzzle, and store it in our cache. The contents of the cache are overwritten in a FIFO order when the cache is full. We then sample a batch of states S from the cache, compute the argmax neighbor values according to the auxiliary target network $V_{\tilde{\theta}}$ for each sampled state, and minimize the difference between the current value and the one-step look ahead value (line 10). The main challenge in Algorithm 3 is lies in constructing the irreps matrices $\rho(s)$ in lines 5-6. We will see how to construct these irreps in Chapter 11.

Algorithm 3: Value Iteration in Fourier Space

Result: V_θ

Input: num epochs T, set of irreps I, batchsize B, max cache size C, target update interval K, random walk length l, discount γ

for $t \leftarrow 1$ **to** T **do**
- Sample a random walk of length l: $(s_1, s_2, \dots s_l)$
- Store $(s_1, s_2, ..., s_l)$ in circular cache
- Sample a batch of states S from cache
- $S_t = \texttt{stack}$ ([$\texttt{vec}(\rho(s))$ for all $s \in S$, $\rho \in I$])
- $S' = \texttt{stack}$ ([$\texttt{vec}(\rho(s'))$ for all s' adjacent to s for $s \in S, \rho \in I$])
- Evaluate $V_\theta(S_t) = \theta^\top S_t$; $V_{\tilde{\theta}}(S') = \tilde{\theta}^\top S'$
- Let S_{t+1} be the argmax neighbors of each $s \in S_t$ from $V_{\tilde{\theta}}(S')$
- $R_t = \texttt{stack}$([1 if s is solved -1 for all $s \in S_{t+1}$])
- $L(\theta) = (V_\theta(S_t) - (R_t + \gamma V_{\tilde{\theta}}(S_{t+1})))^2$
- Update θ with $\nabla_\theta L(\theta)$
- **if** $t \mod K == 0$ **then**
  - $\tilde{\theta} \leftarrow \theta$

end for

CHAPTER 11

CONSTRUCTING THE FOURIER BASIS

In Chapter 10, we set out to represent a value function $V : \mathcal{G} \to \mathbb{C}$ in the Fourier (irrep) basis. This value function would then be expressed as a linear combination of the irrep basis functions:

$$V(g) = \sum_{\rho \in I} \mathrm{Tr}[W_\rho^t op\rho(g)]$$

Or alternatively, if we rewrite the trace as a linear expansion of the matrix elements, we have:

$$V(g) = \sum_{\rho \in I} \sum_{ij} [W_\rho]_{ij} [\rho(g)]_{ij}.$$

These value functions are just linear functions of the matrix entries of the chosen irreps so the main challenge lies in computing the irrep matrices. For a given group element g of our chosen group, we need to evaluate $\rho(g)$ for all ρ in the chosen set of irreps that we are using to represent the value function. It turns out that computing these irreducible representations of the 2-by-2 cube group will require us to first construct irreps of symmetric groups of various orders, irreps of C_3^8 and construct so-called induced representations. Fortunately, there exist known prescriptions for computing irreps of $\mathbb{S}_n$. Young's Orthogonal Representation (YOR) is one such explicit way of constructing the irreps of the symmetric group. Our description of YOR largely follows the presentation given by Huang et al. [2009], and Kondor [2008].

11.1 Preliminaries

Recall from Part I, that a compact way of denoting permutations of $\mathbb{S}_n$ is through **cycle notation**. Permutations can be expressed as a product of disjoint cycles. The cycle $(a_1, \ldots a_k)$ denotes the permutation that sends $a_1 \mapsto a_2, a_2 \mapsto a_3, \ldots, a_k \mapsto a_1$. Without

loss of generality, we can omit singleton cycles going forward.

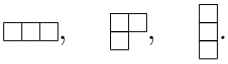
The representations of $\mathbb{S}_n$ are indexed by **partitions** of n, a set of positive integers that sum to n. The tuple of non-negative integers $\lambda = (\lambda_1, \lambda_2 \ldots \lambda_k)$ is a partition of n if $\sum_{i=1}^{k} \lambda_i = n$. Conventionally, the parts of a partition are listed in weakly decreasing order: $\lambda_1 \geq \ldots \geq \lambda_k$. We use the notation $\lambda \vdash n$ to indicate that λ is a partition of n.

Partitions can be visualized by **Ferrers diagrams**, patterns of unfilled left aligned boxes where row i contains λ_i boxes.

Definition 11.1.1. *A **Young tableau** is a assignment of the numbers $1, 2, \ldots n$ to the n boxes of a Ferrers diagram.*

Definition 11.1.2. *A **Young standard tableau** is a Young tableau where the entries are strictly increasing along each row and across each column.*

For convenience, we will also refer to Young standard tableaux as standard tableaux. The partition underlying a Ferrers diagram or a Young tableau as also referred to as its shape.

Example 21. *For $n = 3$, there are three possible partitions $(3), (2,1)$ and $(1,1,1)$, which have the respective Ferrers diagrams:*  .

Example 22. *There are only two possible Young standard tableaux with the shape $(2,1)$:* $\begin{array}{|c|c|}\hline 1 & 2 \\ \hline 3 \\ \cline{1-1}\end{array}$ *and* $\begin{array}{|c|c|}\hline 1 & 3 \\ \hline 2 \\ \cline{1-1}\end{array}$.

Permutations act on the set of Young standard tableaux by permuting the entries of the boxes. For $\sigma \in \mathbb{S}_n$ and t a standard tableau, the action of σ on t is denoted $\sigma \circ t$ and is given by permuting the entries of t by σ. For example, $(2,3) \in \mathbb{S}_3$. The action of $(2,3)$ on the standard tableau $\begin{array}{|c|c|}\hline 1 & 3 \\ \hline 2 \\ \cline{1-1}\end{array}$ is

$$(2,3) \circ \begin{array}{|c|c|}\hline 1 & 3 \\ \hline 2 \\ \cline{1-1}\end{array} = \begin{array}{|c|c|}\hline 1 & 2 \\ \hline 3 \\ \cline{1-1}\end{array}$$

Definition 11.1.3. *The **axial distance** $d_t(i,j)$ between entries i and j in tableau t is defined to be*

$$d_t(i,j) = (col(t,j) - col(t,i)) - (row(t,j) - row(t,i))$$

where $col(t,i)$ is the column index of label i in tableau t and similarly $row(t,i)$ is the row index of label i in tableau t. Also, $d_t(i,j) = -d_t(j,i)$

Example 23. *For* $t = \begin{array}{|c|c|}\hline 1 & 3 \\ \hline 2 \\ \cline{1-1}\end{array}$*, we have the axial distances:*

$$d_t(1,2) = (1-1) - (2-1) = -1 \tag{11.1}$$

$$d_t(1,3) = (2-1) - (1-1) = 1 \tag{11.2}$$

$$d_t(2,3) = (2-1) - (1-2) = 2 \tag{11.3}$$

11.2 Defining Young's Orthogonal Representation on Transpositions

Young's Orthogonal Representation is one instantiation of the irreps irreps of $\mathbb{S}_n$ that is relatively efficient to compute. Given a set of generators for $\mathbb{S}_n$, it suffices to define the irrep matrices on all generator elements. Then any $\sigma \in \mathbb{S}_n$ can be expressed as a product of these generators: $\sigma = g_1 g_2 \dots g_k$ and using the property of irrep matrices we can produce $\rho(\sigma)$ from $\rho(\sigma) = \rho(\sigma_1) \dots \rho(\sigma_k)$.

A **transposition** is a permutation that swaps only two elements. Any cycle $(c_1, ..., c_l)$ is the product of transpositions: $(c_1, c_2)(c_2, c_3) \dots (c_{l-1}, c_l)$. Furthermore, any transposition (i,j) can be written as a product of adjacent transpositions of the form $\tau_k = (k, k+1)$

$$(i,j) = \tau_{j-1}, \dots \tau_{i+1}\tau_i\tau_{i+1} \dots \tau_{j-1}$$

The set of transpositions: $(1,2), (2,3), \dots (n-1, n)$ generate $\mathbb{S}_n$, so we only need to define Young's Orthogonal Representation for these transpositions.

For partition λ, we we denote the Young Orthogonal Representation associated with λ as $\rho_\lambda : \mathbb{S}_n \to \mathbb{C}^{d_\lambda \times d_\lambda}$, where d_λ is the dimension of ρ_λ. It turns out that d_λ is also the number

of standard tableaux of shape λ.

Suppose we are given a fixed ordering of the standard Young tableaux of shape λ: $t_1, t_2, \ldots, t_{d_\lambda}$. We will refer to the rows and columns of the ρ_λ matrices using the standard tableau in the fixed ordering above. The first standard tableau t_1 refers to the first column and row, t_2 refers to the second column and row, etc. $[\rho_\lambda(\sigma)]_{t_i,t_j}$ refers to the (i,j) entry of $\rho_\lambda(\sigma)$. For a transposition $(i-1,i)$, the entries of the YOR matrix $\rho_\lambda(i-1,i)$ are defined by the following rules:

1. On the diagonal entries:
$$[\rho_\lambda(i-1,i)]_{tt} = \frac{1}{d_t(i-1,i)}$$

2. For all entries (t_j, t_k) where $t_k \neq (i-1,i) \circ t_j$:
$$[\rho_\lambda(i-1,i)]_{t_j,t_k} = 0$$

3. If $(i-1,i) \circ t_j = t_k$, then
$$[\rho_\lambda(i-1,i)]_{t_j,t_k} = \sqrt{1 - \frac{1}{d^2_{t_j}(i-1,i)}}$$

Now that we have a prescription for computing the ρ_λ YOR matrices of any transposition of the form $\tau_k = (k, k+1)$, we can use them to compute representations for any other permutation. For $\sigma \in \mathbb{S}_n$, we first decompose σ into a product of transpositions: $\sigma = \tau_{a_1} \ldots \tau_{a_k}$ using bubblesort as suggested by Huang et al. [2009], Kondor [2008]. Then $\rho_\lambda(\sigma)$ is simply the product of the YOR matrices of σ's constituent transpositions:

$$\sigma = \tau_{a_1} \ldots \tau_{a_k} \tag{11.4}$$

$$\rho_\lambda(\sigma) = \rho_\lambda(\tau_{a_1}) \ldots \rho_\lambda(\tau_{a_k}) \tag{11.5}$$

Example 24. $\mathbb{S}_3$ *is generated by the two transpositions:* $\tau_1 = (1,2), \tau_2 = (2,3)$. *For the partition* $\lambda = (2,1)$ *we have exactly two standard tableaux so* ρ_λ *is a* 2×2 *irrep. The YOR irrep matrices* $\rho_{\boxplus}$ *for the elements of* $\mathbb{S}_3$ *are:*

$$\rho_{\boxplus}(e) = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}, \rho_{\boxplus}(1,2) = \begin{pmatrix} -1 & 0 \\ 0 & 1 \end{pmatrix}, \rho_{\boxplus}(2,3) = \begin{pmatrix} \frac{1}{2} & \frac{\sqrt{3}}{2} \\ \frac{\sqrt{3}}{2} & -\frac{1}{2} \end{pmatrix}$$

$$\rho_{\boxplus}(1,3) = \begin{pmatrix} \frac{1}{2} & -\frac{\sqrt{3}}{2} \\ -\frac{\sqrt{3}}{2} & -\frac{1}{2} \end{pmatrix}, \rho_{\boxplus}(1,2,3) = \begin{pmatrix} -\frac{1}{2} & -\frac{\sqrt{3}}{2} \\ \frac{\sqrt{3}}{2} & -\frac{1}{2} \end{pmatrix}, \rho_{\boxplus}(1,3,2) = \begin{pmatrix} -\frac{1}{2} & \frac{\sqrt{3}}{2} \\ -\frac{\sqrt{3}}{2} & -\frac{1}{2} \end{pmatrix}$$

The last three permutations of $\mathbb{S}_3$ *are:* $(1,3), (1,2,3)$ *and* $(1,3,2)$. *To compute their* $\rho_{\boxplus}$ *matrices, we express each permutation as a product of the transpositions* $(1,3)$ *and* $(2,3)$. *So* $(1,2,3) = (1,2)(2,3)$, *giving us* $\rho_{\boxplus}(1,2,3) = \rho_{\boxplus}(1,2)\rho_{\boxplus}(2,3)$, *and so on for the remaining permutations.*

11.3 Dimensions of the Irreps of the Symmetric Group

The number of standard tableau of shape λ, which is also the dimension of the irrep ρ_λ, is given by the **hook length formula**. Given any box of a Ferrers diagram of shape λ, its corresponding hook is defined to be that box, the boxes in its row to its right and the boxes in its column below it. The hook length of a box is the number of boxes in its hook. The hook length formula is

$$f^\lambda = \frac{n!}{\Pi_i l_i}$$

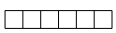
where l_i is the hook length of box i, and i ranges over all boxes of the λ shaped Ferrers diagram.

Example 25. *The Ferrers diagram of shape* $\lambda = (2,1)$ *is* $\begin{array}{|c|c|}\hline a & b \\ \hline c \\ \cline{1-1}\end{array}$. *The hook lengths of the boxes are:* $l_a = 3, l_b = 1, l_c = 1$. *So* $f^{\boxplus} = \frac{3!}{3 \cdot 1 \cdot 1} = 2$.

Example 26. *The Ferrers diagram of shape* $\lambda = (2,1,1)$ *is* $\begin{array}{|c|c|}\hline a & b \\ \hline c \\ \cline{1-1} d \\ \cline{1-1}\end{array}$. *The hook lengths of the boxes are:* $l_a = 4, l_b = 1, l_c = 2, l_d = 1$. *So* $f^{\ydiagram{2,1,1}} = \frac{4!}{4 \cdot 1 \cdot 2 \cdot 1} = 3$.

In Table 11.1, we show the hook length or irrep dimension for a few of the partitions of $n = 6$.

Table 11.1: Dimension of irreps of $\mathbb{S}_6$

λ	Ferrers Diagram	f^λ
(6)		1
$(5,1)$		5
$(4,2)$		9
$(4,1,1)$		10
$(3,3)$		5

11.4 Irreducible Representations of Wreath Product Groups

Now that we know how to construct the irreducible representations of $\mathbb{S}_n$, we are ready to describe the construction of the irreps of cyclic wreath product groups. We follow the constructions given by Kerber [1971] and Rockmore [1995].

11.4.1 Irreducible Representations of C_m^n

Recall that the irreps of C_m are the complex exponentials: $\chi_j(k) = e^{ijk/m}$ for $j = 0, 1, \ldots, m-1$. The irreps of C_m^n are: $\chi_{j_1} \otimes \chi_{j_2} \otimes \ldots \otimes \chi_{j_n}$ for $j_1, ..., j_n \in \{0, \ldots m-1\}$. For $x = (x_1, x_2, \ldots, x_n) \in C_m^n$, where each $x_i \in C_m$, the evaluation of this irrep on x is:

$$(\chi_{j_1} \otimes \chi_{j_2} \otimes \ldots \otimes \chi_{j_n})(x) = \bigotimes_{i=1}^{n} \chi_{j_i}(x_i) = \prod_{i=1}^{n} \chi_{j_i}(x_i) \tag{11.6}$$

The last equality holds because the χ_j irreps are 1-dimensional so the tensor product of the irreps is just a scalar multiplication.

In subsequent sections, for partitions of n into m non-negative parts, $\alpha = (\alpha_1, \alpha_2, \ldots, \alpha_m)$, $\alpha_i \geq 0$, $\sum_{j=1}^{m} \alpha_j = n$, and $x \in C_m^n$, we will use the notation χ^α to denote the irrep of C_m^n that is comprised of

- α_1 copies of the χ_0 irrep applied to the the first α_1 entries of x: $x_1, \ldots, x_{\alpha_1}$,
- α_2 copies of the χ_1 irrep applied to the next α_2 entries of x: $x_{\alpha_1+1}, \ldots x_{\alpha_1+\alpha_2}$,

  $\vdots$
- α_m copies of the χ_{m-1} irrep applied to the final α_m entries of x: $x_{n-\alpha_m+1}, \ldots, x_n$.

Formally, χ^α is the irrep of C_m^n that evaluates as:

$$\chi^\alpha(x) = \underbrace{\chi_0(x_1)\ldots\chi_0(x_{\alpha_1})}_{\alpha_1 \text{ terms}}\underbrace{\chi_1(x_{\alpha_1+1})\ldots\chi_1(x_{\alpha_1+\alpha_2})}_{\alpha_2 \text{ terms}}\cdots\underbrace{\chi_{m-1}(x_{n-\alpha_m+1})\ldots\chi_{m-1}(x_n)}_{\alpha_m \text{ terms}} \tag{11.7}$$

For $f \in C_m^n$, $\pi \in \mathbb{S}_n$, let $f = (f_1, ..., f_n)$. The permutation action of $\mathbb{S}_n$ on the elements of C_m^n, denoted $\pi \circ f$ is a permutation of the the indices of f:

$$\pi \circ f = (f_{\pi^{-1}(1)}, f_{\pi^{-1}(2)}, \ldots, f_{\pi^{-1}(n)}). \tag{11.8}$$

11.4.2 Young Subgroups

The **Young Subgroup** corresponding to a partition $\alpha = (\alpha_1, \ldots, \alpha_m)$, where $\sum_{i=1}^{m} \alpha_i = n$, $\alpha_i \geq 0$ for $i = 1, \ldots m$ is the subgroup of $\mathbb{S}_n$ consisting of permutations that that permute the set $\{1, 2, \ldots, \alpha_1\}$ amongst themselves, $\{\alpha_1, \alpha_1 + 1, \ldots, \alpha_2\}$ amongst themselves, and so

on. We can also express this group using the following notation:

$$\mathbb{S}_\alpha = \mathbb{S}_{\{1,\ldots,\alpha_1\}} \times \mathbb{S}_{\{\alpha_1+1,\ldots,\alpha_1+\alpha_2\}} \times \ldots \times \mathbb{S}_{\{n-\alpha_m+1,\ldots,n\}},$$

It is straightforward to see that $\mathbb{S}_\alpha$ is also isomorphic to $\mathbb{S}_{\alpha_1} \times \mathbb{S}_{\alpha_2} \times \ldots \times \mathbb{S}_{\alpha_m}$. Thus, the irreps of $\mathbb{S}_\alpha$ are the irreps of this product group as well.

Definition 11.4.1 (Young Subgroup Irreps)**.** *The irreps of the Young Subgroup $\mathbb{S}_\alpha$ are constructed by taking the tensor product of irreps of each individual $\mathbb{S}_{\alpha_i}$. Suppose we have $\psi = (\psi_1, \psi_2, ..., \psi_m)$, where each ψ_i is a partition of α_i, then ψ indexes an irrep of $\mathbb{S}_\alpha$ and can be expressed as ρ_ψ:*

$$\rho_\psi = \rho_{\psi_1} \otimes \rho_{\psi_2} \otimes \ldots \otimes \rho_{\psi_m}. \tag{11.9}$$

We can compute each ρ_{ψ_i} using Young's Orthogonal Representation as detailed in Section 11.2.

Example 27. *For $\alpha = (2,2), \mathbb{S}_\alpha = \mathbb{S}_{\{1,2\}} \times \mathbb{S}_{\{3,4\}} = \{e, (1,2), (3,4), (1,2)(3,4)\}$. The irreps of $\mathbb{S}_\alpha$ are the tensor product of the irreps of $\mathbb{S}_2$ by the irreps of $\mathbb{S}_2$: $(\rho_{\ydiagram{2}} \otimes \rho_{\ydiagram{2}}), (\rho_{\ydiagram{2}} \otimes \rho_{\ydiagram{1,1}}), (\rho_{\ydiagram{1,1}} \otimes \rho_{\ydiagram{2}})$, and $(\rho_{\ydiagram{1,1}} \otimes \rho_{\ydiagram{1,1}})$.*

11.4.3 Induced Representations

Given a group $\mathcal{G}$ with a subgroup H. Let $g_1, \ldots g_r$ be a set of left coset representatives of H in G. So $G = g_1 H \cup \ldots \cup g_r H$, with $r = [\mathcal{G} : H]$. $[\mathcal{G} : H]$ is also known as the coset index of H in G and is equal to: $[\mathcal{G} : H] = \frac{|\mathcal{G}|}{|H|}$. Suppose H has the representation $\rho : H \to \mathbb{C}^{n \times n}$. Then the **induced representation** $\tilde{\rho} : \mathcal{G} \to \mathbb{C}^{nr \times nr}$ is defined as the representation with

the following evaluation:

$$\tilde{\rho}(x) = \begin{pmatrix} \rho(g_1^{-1}xg_1) & \rho(g_1^{-1}xg_2) & \dots & \rho(g_1^{-1}xg_r) \\ \rho(g_2^{-1}xg_1) & \rho(g_2^{-1}xg_2) & \dots & \rho(g_2^{-1}xg_r) \\ \vdots & \vdots & \ddots & \vdots \\ \rho(g_r^{-1}xg_1) & \rho(g_r^{-1}xg_2) & \dots & \rho(g_r^{-1}xg_r) \end{pmatrix} \tag{11.10}$$

where ρ (a representation of H) is extended to the domain $\mathcal{G}$ by defining $\rho(x) = \mathbf{0}$ (an $n \times n$ zero matrix) if $x \notin H$ in . Sagan [2013], and other introductory texts on representation theory provide standard proofs that this construction of $\tilde{\rho}$ does in fact result in a valid representation of $\mathcal{G}$. We denote the induced representation of ρ from H to $\mathcal{G}$ as $\tilde{\rho} = \text{Ind}_H^{\mathcal{G}}\rho$.

An important property of induced representations is that only one entry of every block row of Eq. (11.4.3) will be non-zero, and likewise for every block column. This means that that the total number of non-zero blocks of $\text{Ind}_H^{\mathcal{G}}\rho$ will be $[\mathcal{G} : H]$. At most, only $\frac{1}{[\mathcal{G}:H]}$ of the entries of $\text{Ind}_H^{\mathcal{G}}\rho(g)$ will be non-zero for all $g \in \mathcal{G}$, giving us a sparse matrix which can be stored in memory more efficiently.

11.4.4 Putting it all together

Up to this point, we have described Young subgroups and how to construct representations of $\mathbb{S}_n$ that are induced from irreps of $\mathbb{S}_\alpha$. We have not yet discussed how irreps of cyclic wreath product groups are actually formed.

It turns out that the irreps of $C_m \wr \mathbb{S}_n$ are indexed by tuples (α, ψ), where α is a partition of n into m non-negative parts: $\alpha = (\alpha_1, \dots, \alpha_m)$, and $\psi = (\psi_1, \dots, \psi_m)$ is a tuple of m partitions, where ψ_i is a positive partition of α_i. Some of the α_i's may be 0, but each of the ψ_j partitions must only contain positive integers. Going forward, we will denote the irrep of $C_m \wr \mathbb{S}_n$ indexed by (α, ψ) by $\rho_{(\alpha,\psi)}$. As we will see shortly, the (α, ψ) irrep of $C_m \wr \mathbb{S}_n$ will be formed by taking the ρ_ψ representation of $\mathbb{S}_\alpha$ (Equation 11.4.1), inducing it up to $\mathbb{S}_n$ and

applying a slight modification to the block matrices of $\text{Ind}_{\mathbb{S}_\alpha}^{\mathbb{S}_n}\rho_\psi$.

Example 28. *The tuple* (α, ψ)*, where* $\alpha = (2,3,3)$*,* $\psi = ((2),(1,1,1),(2,1))$*, indexes an irrep of* $C_3 \wr \mathbb{S}_8$*.*

Let a full set of coset representatives of the Young Subgroup $\mathbb{S}_\alpha$ in $\mathbb{S}_n$ be $R = \{g_1, ..., g_k\}$. Recall from the definition of a wreath product group (Definition 7.1.4) that a group element in $C_m \wr \mathbb{S}_n$ can be written as a tuple (f, π) where $f \in C_m^n$ and $\pi \in \mathbb{S}_n$. For $(f, \pi) \in C_m \wr \mathbb{S}_n$, the irrep matrix $\rho_{(\alpha,\psi)}(f, \pi)$ is the block matrix, with the (i, j) block defined as

$$\begin{aligned}[\rho_{(\alpha,\psi)}(f,\pi)]_{ij} &= \chi^\alpha(g_i^{-1} \cdot f) \cdot [\text{Ind}_{\mathbb{S}_\alpha}^{\mathbb{S}_n}\rho_\psi(\pi)]_{ij} \\ &= \chi^\alpha(g_i^{-1} \cdot f) \cdot \rho_\psi(g_i^{-1}\pi g_j)\end{aligned}$$

for $i, j \in \{1, \ldots, k\}$. where:

- the action of $g \in \mathbb{S}_n$ on $f \in C_m^n$, $(g^{-1} \cdot f)$ is defined in Equation (11.8),
- χ^α is defined in Equation (11.7))
- $\rho_\psi = \bigotimes_{i=1}^{m} \rho_{\psi_i}$, is an irrep of the Young Subgroup $\mathbb{S}_\alpha$ as defined in Definition 11.4.1.

The dimension of this representation is $d_{(\alpha,\psi)} = [\mathbb{S}_n : \mathbb{S}_\alpha] \cdot \prod_{i=1}^{m} d_{\psi_i}$, where d_{ψ_i} is the dimension of the irrep ρ_{ψ_i} of $\mathbb{S}_{\alpha_i}$. Recall that C_m is an abelian group, so it only has one dimensional irreps, which also means that $\chi^\alpha(g_i^{-1} \cdot f)$ is one dimensional.

Example 29. *Let* $\alpha = (2,3,3)$ *and* $\psi = ((2),(1,1,1),(1,1,1))$ *for the irrep indexed by* (α, ψ) *of* $C_3 \wr \mathbb{S}_8$*.* $\mathbb{S}_\alpha = \mathbb{S}_2 \times \mathbb{S}_3 \times \mathbb{S}_3$*. The coset index of* $\mathbb{S}_{(2,3,3)}$ *in* $\mathbb{S}_8$ *is:* $[\mathbb{S}_8 : \mathbb{S}_{(2,3,3)}] = \frac{8!}{2!3!3!} = 560$*. The irrep indexed by partition* (2) *of* $\mathbb{S}_2$ *has dimension* 1 *and so does the irrep indexed by* $(1,1,1)$ *of* $\mathbb{S}_3$*. So the dimensionality of irrep* (α, ψ) *is:* $d_{(\alpha,\psi)} = 560 \times 1 \times 1 \times 1 = 560$*.*

Example 30. *Let* $\alpha = (4,2,2)$ *and* $\psi = ((2,2),(1,1),(1,1))$*. The irrep* (ψ, α) *of* $\mathbb{C}_3 \wr \mathbb{S}_8$*:* $[\mathbb{S}_8 : \mathbb{S}_{(4,2,2)}] = \frac{8!}{4!2!2!} = 420$*. Irrep* $(2,2)$ *of* $\mathbb{S}_4$ *has dimension* 4 *and irrep* $(1,1)$ *of* $\mathbb{S}_2$ *has dimension* 1*. So the dimensionality of irrep* $d_{(\alpha,\psi)} = 420 \times 4 \times 1 \times 1 = 1680$*.*

11.5 Implementation Details for Computing Wreath Product Irreps

In the previous section, we provided mathematical formulae for computing the irreps. In this section we will give some insight into how we actually computed these irreps in code.

All of the irreps of $\mathbb{S}_8$ and Pyraminx $(C_2 \wr \mathbb{S}_6)$ can be computed once and loaded into RAM when needed since the size of the groups are relatively small. We do this by constructing a dictionary mapping group elements which can be encoded as tuples of integers to their associated representation matrix for every single irrep. The 2-by-2 cube group $(C_3 \wr \mathbb{S}_8)$, however, is too large to load every irrep matrix into RAM so we only store and load the irrep matrices ρ_ψ of $\mathbb{S}_\alpha$ and $\text{Ind}_{\mathbb{S}_\alpha}^{\mathbb{S}_n}\rho_\psi$. At runtime, we compute the scalar factors $\chi^\alpha(g_i^{-1} \cdot f)$ to multiply with each non-zero (i, j) block of $\text{Ind}_{\mathbb{S}_\alpha}^{\mathbb{S}_n}\rho_\psi$ to construct $\rho_{(\alpha,\psi)}$.

To construct the irreps indexed by (α, ψ) of $C_n \wr \mathbb{S}_m$ we need to perform the following computations:

- Compute a set of coset representatives of $\mathbb{S}_\alpha$ in $\mathbb{S}_n$: $R = \{g_1, \ldots, g_k\}$ where $k = [\mathbb{S}_n : \mathbb{S}_\alpha]$. This can be done with mathematical software such as SageMath or a library for symbolic math such as Sympy. We implemented our own permutation class in python which was feasible since $|\mathbb{S}_8| = 40320$ is small enough to brute force.

- Construct the irrep ρ_ψ of Young Subgroup $\mathbb{S}_\alpha$, by taking the tensor product of the Symmetric group irreps: $\rho_\psi(g) = \rho_{\psi_1}(g) \otimes \ldots \otimes \rho_{\psi_m}(g)$. Cache the mapping from group element g to its corresponding irrep matrix $\rho_\psi(g)$ in a dictionary/hashmap and store it to disk as a pickle/binary file

- For each $\pi \in \mathbb{S}_n$, determine the block sparsity pattern of the induced representation $\text{Ind}_{\mathbb{S}_\alpha}^{\mathbb{S}_n}\rho_\psi$, cache the block permutation structure: This just involves computing the (i, j) indices such that $g_i^{-1}\pi g_j \in \mathbb{S}_\alpha$. Again, since we only need to do this for $\mathbb{S}_n = \mathbb{S}_8$,

it is feasible to brute force this computation. Cache the indices for each $\pi \in \mathbb{S}_n$ in a dictionary/hashmap and save it to disk in a pickle/binary file.

- At runtime, load the pickle/binary files of the irrep of $\mathbb{S}_\alpha$ and the nonzero block entries of the induced representation. Evaluate the irrep matrix of $(f, \pi) \in C_m^n \wr \mathbb{S}_n$:
  - load the representations $\rho_\psi(\pi)$ from disk
  - load the non-zero block indices $\{(i,j) \mid 1 \leq i,j \leq k, g_i^{-1}\pi g_j \in \mathbb{S}_\alpha\}$ from disk
  - compute the block scalars: $\chi^\alpha(g_i^{-1} \cdot f)$
  - construct the block matrix using the cached values, which effectively amounts to performing table lookups and performing a scalar multiplication on various blocks of the irrep matrix

As mentioned earlier, constructing the irreps of S_α amounts to taking tensor products of the individual irresp of $\mathbb{S}_{\alpha_i}$. The partition parts of α are all between 0 and 8 so in fact, we can enumerate all possible group elements of $\mathbb{S}_\alpha$ and manually compute tensor products of individual group elements of $\mathbb{S}_{\alpha_i}$.

CHAPTER 12

EXPERIMENTS

We demonstrate the efficacy of the learning over the Fourier bases of the following three permutation puzzles using Algorithm 3: Pyraminx (Fig 7.2a), $\mathbb{S}_8$, and the 2-by-2 cube (Fig 7.1).

- Pyraminx (without the tips): a subgroup of $C_2 \wr S_6$ with 11520 states, and a diameter of 9. The tips can trivially be moved to the correct position, so we choose to ignore them.
- An $\mathbb{S}_8$ puzzle with 40320 states and a diameter of 9. The generators of this puzzle are the $\mathbb{S}_8$ permutations associated with the six valid face moves of the 2-by-2 cube.
- 2-by-2 Rubik's Cube: a subgroup of $C_3 \wr \mathbb{S}_8$ with 3.67×10^6 states, and a diameter of 14. Note that we are using a symmetrized version of the 2-by-2 cube, by modding out the 24 rotational symmetries of the cube (the full 2-by-2 cube has 88 million states).

The goal of our experiments is to show that we can learn effective value functions in the Fourier basis , where effectiveness is measured according to:

1. How often does our learned value function give a locally optimal move at a given state? When evaluating V_θ over all neighbors of a given state, is the argmax neighbor in fact closer to the goal state?
2. If we use the value network in a greedy/best first search manner, how often does this policy reach the goal state?
3. How well does the value function work when used in heuristic search?

We compare our models, which we will refer to as Fourier or irrep models going forward, against deep value networks (DVN) and an optimal solver, which simply uses the ground

truth shortest path distance as the value function. These three puzzles are small enough that we can use Djikstra's algorithm compute the shortest path distance of each state to the goal state which is also used to compute the proportion of locally optimal moves. The DVN uses onehot encoding representations of the puzzles while the Fourier models use the Fourier basis representations.

12.1 Baseline DVN

For Pyraminx and the $\mathbb{S}_8$ puzzle, we parameterize the DVN as a multilayer perceptron with one hidden layer and rectified linear unit nonlinearities; the hidden layer sizes were 1024 and 2048 respectively. For the 2-by-2 cube, the DVN architecture follows a similar architecture to the one used by Agostinelli et al. [2019]: five fully connected layers with a rectified linear unit nonlinearity placed after every non-output layer, and a residual connections on the third and fourth fully connected layers. The layers have the following input and output dimensions: (88, 1024), (1024, 2048), (2048, 2048), (2048, 2048), (2048, 1), where 88 is the size of the one-hot encoding of a 2-by-2 cube state. See Figure 12.1 for a visual description of the deep value networks.

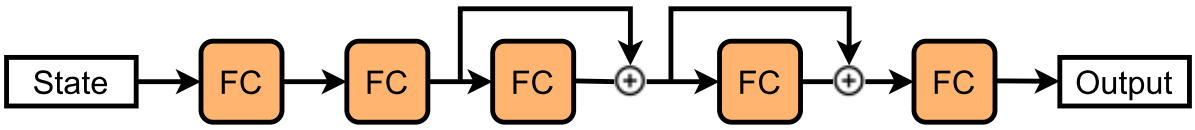


Figure 12.1: Deep Value Network architecture for Pyraminx, $\mathbb{S}_8$ and 2-by-2 cube puzzles. Every fully connected layer (besides the final layer) is followed by a rectified linear unit activation. This DVN architecture is inspired by the architecture used by McAleer et al. [2018] and Agostinelli et al. [2019].

For all puzzles, we decided the number of layers, hidden layer sizes, and residual layers by a grid search over the set of hyperparameters. Similar to Agostinelli et al. [2019], we found that increasing the number of layers did not improve training while wider hidden layers did.

We trained the Pyraminx, $\mathbb{S}_8$, and 2-by-2 cube models for a maximum of 60k, 100k, and

300k epochs. For the 2-by-2 cube experiments, we terminated training for irrep models if they showed no improvement in solves or locally optimal moves within the last 10k epochs.

12.2 Hyperparameters

During training, we sample a random walk of length 15 for Pyraminx, the $\mathbb{S}_8$ puzzles, and a random walk of length 25 for the 2-by-2 cube. All model parameters were optimized using Adam [Kingma and Ba, 2014a]. For all the Fourier models, we used a learning rate of 0.005 and a minibatch size of 128 for the two smaller puzzles and 32 for the 2-by-2 cube. For the DVN, we used a learning rate of 0.003 and a minibatch size of 128. The initial weights of the the DVN were drawn from a normal distribution with mean 0 and standard deviation of $0.03, 0.05$ or 0.1. The capacity of the circular cache was $100,000$ and the target network was updated every 100 epochs. In our experience, most of these hyperparameters (except the model architecture of the DVNs) can be slightly modified without changing the performance of the models too much. We picked the various hyperparameters by randomized grid search: after training DVNs for 10k for the smaller puzzles or 50k epochs for the 2-by-2 cube over various parameter settings, we picked the ones that lead to the highest proportion of locally optimal moves. The one parameter that was fairly important to get right was the discount parameter γ. The Fourier and DVN models were optimal when γ was set to 0.99 or larger and did not converge to an effective value function when set to anything lower than 0.98.

For the Fourier models, the only model parameter to choose is the set of irreps/Fourier basis functions to use for parameterizing V_θ. Finding the best k irreps to use would require performing a cross validation over $O(|R|^k)$ different combinations of irreps, which is not practical. Instead, for each puzzle, for each irrep ρ, we trained a Fourier model using only that single irrep ρ over $50,000$ epochs. We then ranked the irreps by the proportion of locally optimal moves made over a random sample of puzzle states. We suspect that a deeper understanding of the group structure of the puzzles will inform us on how to choose

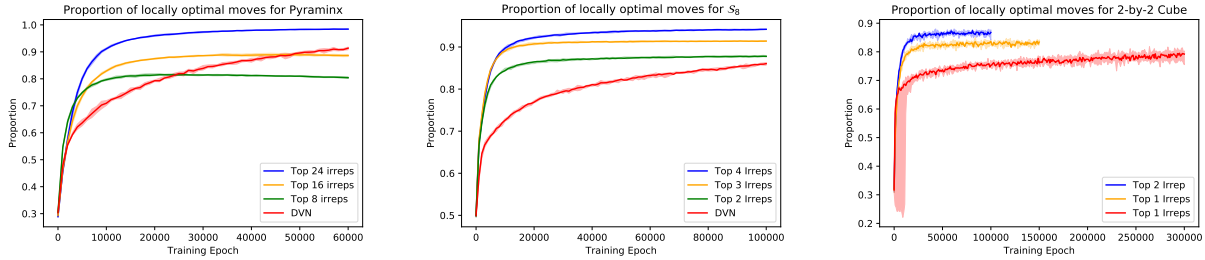
which irreps to use which we leave for future work.

Table 12.1 shows the top ten irreps by the proportion of locally optimal moves predicted and proportion of validation samples that could be solved by following the Fourier model greedily. The results show that the top two irreps are far and away the most effective irreps to use for a set of basis functions to repesent our value function in. For the two smaller puzzles $\mathbb{S}_8$ and Pyraminx, we performed a similar validation procedure to rank the irreps. Since $\mathbb{S}_8$ and $C_2 \wr \mathbb{S}_6$ were much smaller groups, we could afford to use more/most of the irreps in the value function.

Table 12.1: Solve statistics for one irrep Fourier models on 2-by-2 cube after $50,000$ training epochs.

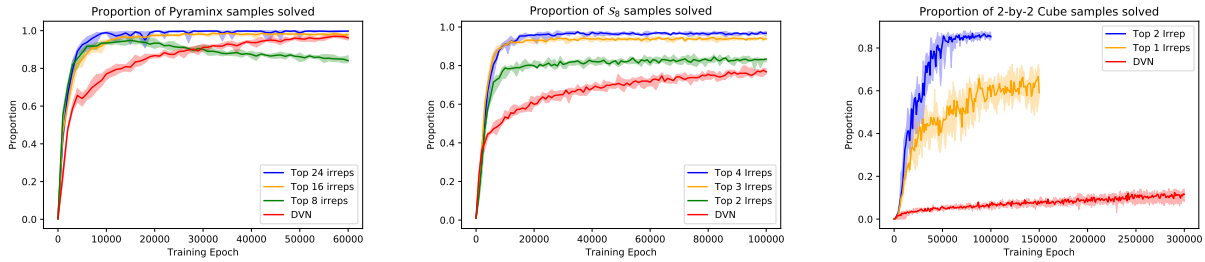
Irrep			
α	ψ	Locally Optimal Moves	Greedy Solves
(2, 3, 3)	((2,), (1,1,1), (1,1,1))	0.83	0.67
(4, 2, 2)	((4,), (1, 1), (1, 1))	0.80	0.44
(4, 2, 2)	((2, 2), (2,), (2,))	0.73	0.07
(2, 3, 3)	((1, 1), (2, 1), (2, 1))	0.71	0.04
(4,2,2)	((4,), (2,), (2,))	0.68	0.02
(2, 3, 3)	((1,1), (2,1), (2,1))	0.66	0.03
(4, 2, 2)	((2,2), (2,), (2,))	0.66	0.02
(2, 3, 3)	((2,), (2,1), (2,1))	0.64	0.04
(3, 1, 4)	((2,1), (1,), (3,1))	0.63	0.03
(3, 4, 1)	((2,1), (3,1), (1,))	0.63	0.03

The algorithm for learning the value function with the DVN is identical to Algorithm 3 except lines $5-6$ use the one-hot encoding of the puzzle states instead of the irreducible representations and V_θ would instead be a neural network parameterized with weights θ. For $\mathbb{S}_n$ the one-hot encoding of $\sigma \in \mathbb{S}_n$ is its corresponding permutation matrix vectorized. For Pyraminx and the 2-by-2 cube, which are both wreath product groups, recall that an element of the wreath product group $C_m \wr \mathbb{S}_n$ can be represented as a tuple (τ, σ), where $\tau \in C_m^n$ and $\sigma \in \mathbb{S}_n$. The onehot encoding of τ is an $m \times n$ binary vector. A onehot encoding of (τ, σ) is just a concatenation of the individual onehot-encodings of τ and σ. So the one-hot



(a) Pyraminx (b) $\mathbb{S}_8$ (c) 2-by-2 Cube

Figure 12.2: Proportion of locally optimal moves



(a) Pyraminx (b) $\mathbb{S}_8$ (c) 2-by-2 Cube

Figure 12.3: Proportion of puzzles solved with greedy policy

encoding of a state of the Pyraminx puzzle $(C_2 \wr \mathbb{S}_6)$ has length $6 \times 6 + 2 \times 6 = 48$. The one-hot encoding of a state from the 2-by-2 cube $(C_3 \wr \mathbb{S}_8)$ has length $8 \times 8 + 3 \times 8 = 88$.

12.3 Evaluation

We evaluated the proportion of local optimal moves made and proportion of random puzzles solved using a greedy policy in conjunction with the learned value function every 1000 epochs. We say that value function produces a locally optimal at state s if the argmax of V on the neighbors of s is in fact a state that is closer to the goal state than s. For the two smaller puzzles, we evaluated the proportion of locally optimal moves and greedy solves over all the legal puzzle states. The 2-by-2 cube is too large to do this for all for all cube states so we instead did this for a random sample of 1000 cubes. We repeated the training for each model

over five random seeds. In Figures 3 and 4, we plot the median values of these metrics with the max and min shaded. All experiments were run on a GeForce GTX 1080 Ti GPU with 64 GB of RAM.

Table 12.2: Proportion of puzzles solved by greedy search

Puzzle	Model	Parameters	% Greedy Solves (± std)	% States seen	Train time (hrs)
Pyraminx	DVN	1.1×10^6	96.2 ± 0.7	100	2.5
	Top 8 Irrep	9.5×10^3	65.4 ± 1.7	100	1.1
	Top 16 Irrep	1.4×10^4	84.1 ± 1.4	100	1.2
	Top 24 Irrep	1.7×10^4	96.9 ± 0.7	100	1.3
$\mathbb{S}_8$	DVN	4.3×10^6	77.0 ± 0.9	93.8	2.9
	Top 2 Irrep	8.0×10^3	81.6 ± 0.1	93.8	1.6
	Top 3 Irrep	9.2×10^3	93.1 ± 0.7	93.8	1.6
	Top 4 Irrep	1.2×10^4	95.2 ± 1.5	93.8	1.7
2-by-2 Cube	DVN	1.0×10^7	11.4 ± 2.1	47.6	14.5
	Top 1 Irrep	6.3×10^5	67.3 ± 2.2	19.2	9.2
	Top 2 Irrep	9.8×10^5	86.6 ± 0.7	13.2	12.3
	Top 2 Irrep - Rank 1	2.8×10^5	9.4 ± 1.1	4.5	2.9
	Top 2 Irrep - Rank 10	2.8×10^4	47.7 ± 1.4	4.5	3.0
	Top 2 Irrep - Rank 100	2.8×10^3	58.8 ± 1.3	4.5	3.5
All Puzzles	Opt Solver(Djikstras)	-	100	100%	-

12.4 Heuristic search with the value function

To get another measure of how effective these learned value functions were, we used the Fourier models and the DVN in A^* search and kept track of the number of states explored. We only do this for the 2-by-2 cube since a greedy/best first search policy is already quite effective on the smaller puzzles. A^* search results in an optimal path if the heuristic function used is *admissable*, which means it never underestimates the true distance to the goal state. If a heuristic function is not admissable, the A^* search will end up searching all the legal states of the puzzle. We have no such admissability guarantees on the Fourier value function or the DVN, but we can still use A^* search since the 2-by-2 is small enough that the search will eventually terminate after visiting all states. We generate 1000 sample test cubes uniformly and tested the Fourier models (including the low rank models) and the DVN. A puzzle state

is considered "explored" after we evaluate the value function on each of its neighbors and add them on the priority queue of puzzle states to explore next. In general, it is preferable to have a heuristic function that requires fewer state explorations. Node statistics using an optimal heuristic function (Djikstra's shortest path) are also shown so that we have a sense of how suboptimal these value functions performed.

12.5 Learning low rank Fourier models

One of the downsides of the Fourier approach is its memory footprint. The top 2 irreps of the 2-by-2 cube have dimensionality 560×560 and 420×420. So the learned Fourier matrices have: $2 \times (560^2 + 420^2) = 9.8 \times 10^5$ parameters. However, as proposed in Section 10.1, we can address the scaling issues by learning low rank Fourier matrices. To demonstrate the feasibility of learning low rank Fourier matrices, we train rank $1, 10$ and 100 Fourier matrices for a Fourier model that uses the top 2 irreducible representation of the 2-by-2 cube. We use these resulting low rank Fourier models in A^*-search as well. Figure 12.4 shows the proportion of greedy solves and proportion of locally optimal moves of the low rank models compared to the full rank model. While the full rank model still performs better than any of the low rank models, the results demonstrate that if we are in a low-parameter setting, we can in fact achieve most of our desired performance with low rank models. Moreover, these low rank models (rank 10 and 100), still outperform the baseline DVN.

12.6 Discussion

Figures 12.2 and 12.3 show that our Fourier approach unequivocally outperforms DVN in terms of the number of states that need to be seen to learn a successful policy. Table 12.2 gives the final proportion of random puzzles solved using each of the learned value functions with greedy search and the number of unique puzzle states seen during training. For the

Table 12.3: 2-by-2 cube A^* search solve statistics

		States explored		
Model	Solved	Lower Quartile	Median	Upper Quartile
Opt Solver(Djikstras)	100%	10	11	11
DVN	100%	25	67.5	179.75
Top 1 Irrep - Full Rank	100%	12	13	16
Top 2 Irrep - Full Rank	100%	11	12	14
Top 2 Irrep - Rank 100	100%	11	13	16
Top 2 Irrep - Rank 10	100%	12	14	17
Top 2 Irrep - Rank 1	100%	13	18	25

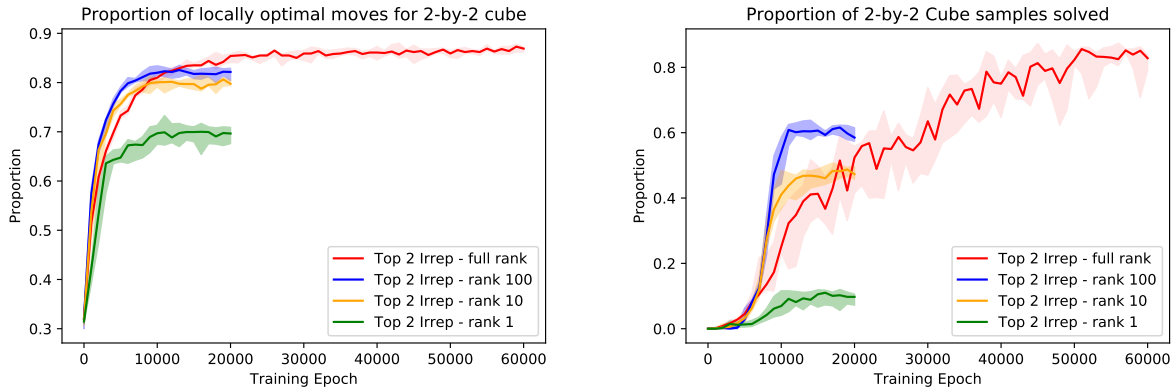
smaller puzzles, the DVN is still a reasonable policy and with more training would likely outperform the Fourier models. For the 2-by-2 cube, the difference between the DVN and the Fourier models is substantial: the Fourier models learn can solve up to 86% of the sampled cubes while the DVN can only solve around 14% at most under a greedy policy. It is not entirely surprising that the Fourier models would outperform the DVN since the DVN must learn a representation from the one-hot encoding of puzzle states. As shown in Table 12.3, when the Fourier based value functions are used in heuristic search, we solve all the randomly sampled test cubes within the allotted state exploration budget. The irrep models do not find optimal (shortest) paths to the solved state (in terms of number of states explored), but considering the number of unique cube states they were trained on (especially the low rank models), we believe the performance is noteable.

Our results lend support to the common complaint that deep reinforcement learning is quite sample inefficient. Using one-hot encodings and deep neural networks as function approximators for solving permutation puzzles is attractive because they avoid the need for any domain expertise; the dynamics of the puzzle can be learned by sufficiently sampling the state space. We show that there is a middle ground between using domain knowledge and learning from the sampled puzzle states through reinforcement learning by using the Fourier basis. We can still learn effective value functions through the same value iteration

techniques and we can do so much more efficiently in the Fourier basis, which are intrinsic to each puzzle.

12.6.1 Miscellaneous Experiment Details

We trained low rank Fourier models in the basis spanned by the top 2 irreps for the 2-by-2 cube. The top two irreps comprise a total of: $560 \times 560 + 420 \times 420 = 490000$ basis functions, which happens to be a perfect square: 700×700. This allowed us to express the parameters of the low rank value function as $\theta_k = U_k V_k^\top \in \mathbb{C}^{700 \times 700}$: where $U_k \in \mathbb{C}^{700 \times k}$, $V_k \in \mathbb{C}^{700 \times k}$ for rank $k = 1, 10, 100$. We trained the low rank models until their performance on proportion of greedy solves and locally optimal moves converged, which happened after 20k epochs. Over the course of these 20k epochs, the low rank models saw only 165k unique cube states, or about 4.5% of all possible states. Figure 12.4 shows the proportion of locally optimal moves and proportion of greedy solves made by the low rank models during training plotted against the performance of the full rank top 2 irrep model. As we can see from these plots, the low rank models have limited capacity but also converge in a fraction of the training epochs required for the full rank model.



(a) Locally optimal moves

(b) Greedy solves

Figure 12.4: Training curve of low rank models for 2-by-2 cube

12.6.2 Hyperparameters

As we mentioned earlier in Section 12.2, we chose the hyperparameters of our training procedure and baseline DVN architecture by doing a randomized grid search over the parameters listed in Table 12.4. While McAleer et al. [2018] and Agostinelli et al. [2019] both used batch normalization layers after each layer's ReLU activation, we found that it slowed training time without improving the proportion of solves and locally optimal moves.

Table 12.4: Hyperparameters

Parameter	Values
seed	{0, 1, 2, 3, 4}
minibatch size	{32, 64, 128}
learning rate	[0.003, 0.006]
discount γ	{0.95, 0.96, 0.97, 0.98, 0.99, 1}
hidden layer size	{512, 1024, 1536, 2048}
number of layers	{1, 2, 3}
weight initialization std	0.03, 0.05, 0.1
replay buffer capacity	{10000, 100000}
target update interval	{50, 100}

12.6.3 Irreps

Tables 12.5, 12.6 and 12.7 show the irreps used for the Pyraminx, $\mathbb{S}_8$ and 2-by-2 cube experiments respectively. The irreps of $\mathbb{S}_n$ are indexed by partitions of n. The irreps of wreath product groups of the form $C_m \wr \mathbb{S}_n$ are indexed by tuples (α, ψ), where α is a partition of n and ψ is a tuple of partitions, where each $\psi_i \in \psi$ is a partition of α_i.

Table 12.5: Top 24 irreps used for Pyraminx

α	ψ	$d_{(\alpha,\psi)}$	$d^2_{(\alpha,\psi)}$
(3, 3)	((2, 1), (3))	40	1600
(3, 3)	((2, 1), (1, 1, 1))	40	1600
(2, 4)	((1, 1), (2, 1, 1))	45	2025
(2, 4)	((2), (3, 1))	45	2025
(4, 2)	((1, 1, 1, 1), (2))	15	225
(2, 4)	((1, 1), (4))	15	225
(1, 5)	((1), (3, 1, 1))	36	1296
(5, 1)	((4, 1), (1))	24	576
(2, 4)	((1, 1), (3, 1))	45	2025
(4, 2)	((1, 1, 1, 1), (1, 1))	15	225
(4, 2)	((2, 1, 1), (2))	45	2025
(0, 6)	((), (5, 1))	5	25
(6, 0)	((1, 1, 1, 1, 1, 1), ())	1	1
(6, 0)	((4, 1, 1), ())	10	100
(6, 0)	((2, 2, 2), ())	5	25
(6, 0)	((3, 3), ())	5	25
(0, 6)	((), (2, 1, 1, 1, 1))	5	25
(1, 5)	((1), (5))	6	36
(1, 5)	((1), (2, 1, 1, 1))	24	576
(1, 5)	((1), (1, 1, 1, 1, 1))	6	36
(1, 5)	((1), (2, 2, 1))	30	900
(1, 5)	((1), (3, 2))	30	900
(2, 4)	((2), (4,))	15	225
(2, 4)	((2), (2, 2))	30	900
		Total basis functions	17621

Table 12.6: Top 4 irreps used for $\mathbb{S}_8$

λ	d_λ	d^2_λ
(4, 2, 2)	56	3136
(3, 2, 2, 1)	70	4900
(5, 1, 1, 1)	35	1225
(3, 3, 1, 1)	56	3136
	Total basis functions	12397

Table 12.7: Top 2 irreps used for the 2-by-2 Cube

α	ψ	$d_{(\alpha,\psi)}$	$d^2_{(\alpha,\psi)}$
(2, 3, 3)	((2), (1,1,1), (1,1,1))	560	313600
(4, 2, 2)	((4), (1,1), (1,1))	420	176400
		Total basis functions	490000

CHAPTER 13

CONCLUSION

13.1 Concluding Remarks

We developed the first approach to using the irreducible representations of wreath product groups to solving the 2-by-2 cube in the reinforcement learning setting. We demonstrated that our Fourier approach can also be further approximated in a low-rank setting. Our method learns a more effective policy for solving the 2-by-2 cube using far fewer samples than the existing standard deep reinforcement learning method. As far as we are aware, this is also the first application of the representation theory of wreath product groups and the symmetric group in reinforcement learning.

13.2 Future Work

There are a few natural questions that our findings bring up:

1. How do we pick the irreps to learn our value function over?
2. Is there a good reason we should be able to learn low rank Fourier matrices?
3. How might we scale these linear Fourier models to larger groups?
4. Is there some way we can combine these Fourier ideas with neural network approaches?

Recall that in our experiments, we first trained individual models over each irrep separately over a limited number of training examples and ranked the irreps based on their performance. Using these rankings, we picked the top k irreps for longer training runs. This is potentially very expensive for larger (non-abelian) groups as both the number and dimension of the irreps will increase with the group size. We were also somewhat lucky that the value function for the 2-by-2 cube could be so well approximated with only 2 irreps. It is

not entirely clear to us why 2 irreps sufficed for the 2-by-2 cube whereas Pyraminx required much more than 2.

At the moment, we do not know for sure why we might be able to expect the learned Fourier matrices to be low rank. One potential line of thought is that we can maybe approximate the true Fourier transform as a random linear combination of the representation matrices and potentially use results from random matrix theory to give us some justification for the low rank findings.

While there are concerns on how well this method can scale, the effectiveness of the low rank Fourier models gives us hope that we can extend this work to larger groups, including the 3-by-3 Rubik's Cube. The 3-by-3 Rubik's cube's group is a subgroup of $(C_3 \wr \mathbb{S}_8) \times (C_2 \wr \mathbb{S}_{12})$. Its irreducible representations are the tensor products $\rho \otimes \phi$, where ρ is an irrep of $C_3 \wr \mathbb{S}_8$ and ϕ is an irrep of $C_2 \wr \mathbb{S}_{12}$. The resulting Fourier matrix has size $(d_\rho \times d_\phi) \times (d_\rho \times d_\phi)$. To learn a value function V for the Rubik's Cube group, we would likely parameterize the learnable Fourier matrices in V as the tensor products of low rank matrices to make the problem tractable. Another potential saving grace is that these irreps of wreath product groups are quite sparse.

Recall that pattern database heuristics solve easier sub-problems of our puzzle and use the solutions to the easier sub-problems as a proxy for solving the original problem. In permutation puzzles where the state space forms a group, a natural sub-problem to consider are states that form a subgroup of the original group. If the puzzle's underlying group is too large to explicitly construct its irreps, is it feasible to construct these Fourier models for solving the puzzles generated by certain subgroups? If so, is there a good way of finding the useful subgroups to consider? In the 2-by-2 cube and the full Rubik's cube, we know that the 2-by-2 cube is in fact a subgroup of the full Rubik's Cube (the subgroup formed by ignoring the edge facets and only considering the corner cubies).

Another promising future direction might also be to use more domain knowledge in neural

network architectures for value functions on permutation puzzles. We saw that the neural networks proposed by McAleer et al. [2018] and Agostinelli et al. [2019] just encoded puzzle states using one-hot encodings and did not exploit any information about the Rubik's cube's symmetry group in their architecture or training procedure. Is there an input representation besides the one-hot encoding and the Fourier/irrep basis encodings of the puzzle states that capture the group structure? Can group theoretic ideas better inform the way we sample our data, parameterize linear/nonlinear functions on the group? Recall that this notion of smoothness for our value function is really a geometric idea that tries to leverage the graph structure of the puzzle's state space in the value function. Are there other properties of permutation groups provide additional structure for our value function?

Finally, we have seen a wealth of research in equivariance in neural networks. It seems natural to consider whether it makes sense to construct equivariant architectures for these permutation puzzles. What do these group equivariant architectures look like in this domain? We leave these open questions to enterprising future researchers.

REFERENCES

Boosted jet identification using particle candidates and deep neural networks. Nov 2017. URL `https://cds.cern.ch/record/2295725`.

Thistlethwaite's 52-move algorithm. `https://www.jaapsch.net/puzzles/thistle.html`, n.d. [Online: Accessed: 03/15/2020].

Forest Agostinelli, Stephen McAleer, Alexander Shmakov, and Pierre Baldi. Solving the Rubik's cube with deep reinforcement learning and search. *Nature Machine Intelligence*, 1(8):356–363, 2019.

Brandon Anderson, Truong Son Hy, and Risi Kondor. Cormorant: Covariant molecular neural networks. *Advances in Neural Information Processing Systems*, 32:14537–14546, 2019.

Shahab Arfaee, Sandra Zilles, and Robert Holte. Learning heuristic functions for large state spaces. *Artif. Intell.*, 175:2075–2098, 10 2011. doi: 10.1016/j.artint.2011.08.001.

Waiss Azizian et al. Expressive power of invariant and equivariant graph neural networks. In *International Conference on Learning Representations*, 2020.

Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. Neural machine translation by jointly learning to align and translate. *arXiv preprint arXiv:1409.0473*, 2014.

Dzmitry Bahdanau, Kyung Hyun Cho, and Yoshua Bengio. Neural machine translation by jointly learning to align and translate. January 2015. 3rd International Conference on Learning Representations, ICLR 2015 ; Conference date: 07-05-2015 Through 09-05-2015.

Song Bai, Feihu Zhang, and Philip HS Torr. Hypergraph convolution and hypergraph attention. *Pattern Recognition*, 110:107637, 2021.

Ivana Balažević, Carl Allen, and Timothy M Hospedales. Tucker: Tensor factorization for knowledge graph completion. *arXiv preprint arXiv:1901.09590*, 2019.

Peter W Battaglia, Jessica B Hamrick, Victor Bapst, Alvaro Sanchez-Gonzalez, Vinicius Zambaldi, Mateusz Malinowski, Andrea Tacchetti, David Raposo, Adam Santoro, Ryan Faulkner, et al. Relational inductive biases, deep learning, and graph networks. *arXiv preprint arXiv:1806.01261*, 2018.

Simon Batzner, Albert Musaelian, Lixin Sun, Mario Geiger, Jonathan P Mailoa, Mordechai Kornbluth, Nicola Molinari, Tess E Smidt, and Boris Kozinsky. E (3)-equivariant graph neural networks for data-efficient and accurate interatomic potentials. *Nature communications*, 13(1):1–11, 2022.

Richard Bellman. A Markovian decision process. *Journal of mathematics and mechanics*, pages 679–684, 1957.

Yoshua Bengio and Yann LeCun. Scaling learning algorithms towards AI. In *Large Scale Kernel Machines*. MIT Press, 2007.

Dimitri P. Bertsekas and John Tsitsiklis. *Neuro-dynamic Programming*. Optimization and Neural Computation Series. Athena Scientific, 1 edition, 1996.

Bastian Bischoff, Duy Nguyen-Tuong, Heiner Markert, and Alois Knoll. Solving the 15-puzzle game using local value-iteration. In *SCAI*, pages 45–54, 2013.

Alexander Bogatskiy, Brandon Anderson, Jan Offermann, Marwah Roussi, David Miller, and Risi Kondor. Lorentz group equivariant neural network for particle physics. In *International Conference on Machine Learning*, pages 992–1002. PMLR, 2020.

Antoine Bordes, Jason Weston, Ronan Collobert, and Yoshua Bengio. Learning structured embeddings of knowledge bases. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 25, 2011.

Antoine Bordes, Nicolas Usunier, Alberto Garcia-Durán, Jason Weston, and Oksana Yakhnenko. Translating embeddings for modeling multi-relational data. In *Proceedings of the 26th International Conference on Neural Information Processing Systems-Volume 2*, pages 2787–2795, 2013.

Joan Bruna, Wojciech Zaremba, Arthur Szlam, and Yann LeCun. Spectral networks and locally connected networks on graphs. *arXiv preprint arXiv:1312.6203*, 2013.

Robert Brunetto and Otakar Trunda. Deep heuristic-learning in the rubik's cube domain: An experimental evaluation. In *ITAT*, pages 57–64, 2017.

Zhaoxing Bu and Richard E Korf. A*+ ida*: A simple hybrid search algorithm. In *IJCAI*, pages 1206–1212, 2019.

Chen Cai and Yusu Wang. Convergence of invariant graph networks. *arXiv preprint arXiv:2201.10129*, 2022.

Tullio Ceccherini-Silberstein, Fabio Scarabotti, and Filippo Tolli. *Representation theory and harmonic analysis of wreath products of finite groups*, volume 410. Cambridge University Press, 2014.

Kyunghyun Cho, Bart Van Merriënboer, Dzmitry Bahdanau, and Yoshua Bengio. On the properties of neural machine translation: Encoder-decoder approaches. *arXiv preprint arXiv:1409.1259*, 2014.

Fan RK Chung and Fan Chung Graham. *Spectral graph theory*. Number 92. American Mathematical Soc., 1997.

Taco Cohen and Max Welling. Group equivariant convolutional networks. In Maria Florina Balcan and Kilian Q. Weinberger, editors, *Proceedings of The 33rd International*

Conference on Machine Learning, volume 48 of *Proceedings of Machine Learning Research*, pages 2990–2999, New York, New York, USA, 20–22 Jun 2016. PMLR. URL `https://proceedings.mlr.press/v48/cohenc16.html`.

Taco S Cohen, Mario Geiger, Jonas Köhler, and Max Welling. Spherical cnns. *arXiv preprint arXiv:1801.10130*, 2018.

Murat Cokol, Nurdan Kuru, Ece Bicak, Jonah Larkins-Ford, and Bree B Aldridge. Efficient measurement and factorization of high-order drug interactions in mycobacterium tuberculosis. *Science advances*, 3(10), 2017.

Michaël Defferrard, Xavier Bresson, and Pierre Vandergheynst. Convolutional neural networks on graphs with fast localized spectral filtering. *Advances in neural information processing systems*, 29:3844–3852, 2016.

Tim Dettmers, Pasquale Minervini, Pontus Stenetorp, and Sebastian Riedel. Convolutional 2d knowledge graph embeddings. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 32, 2018.

Persi Diaconis. *Group Representations in Probability and Statistics*, volume 11 of *Lecture Notes-Monograph Series*. Institute of Mathematical Sciences, 1988.

Matthew J Dolan and Ayodele Ore. Equivariant energy flow networks for jet tagging. *Physical Review D*, 103(7):074022, 2021.

Carmel Domshlak, Michael Katz, and Alexander Shleyfman. Enhanced symmetry breaking in cost-optimal planning as forward search. In *Twenty-Second International Conference on Automated Planning and Scheduling*, 2012.

Yihe Dong, Will Sawin, and Yoshua Bengio. Hnhn: Hypergraph networks with hyperedge neurons. *arXiv preprint arXiv:2006.12278*, 2020.

Alexey Dosovitskiy, Lucas Beyer, Alexander Kolesnikov, Dirk Weissenborn, Xiaohua Zhai, Thomas Unterthiner, Mostafa Dehghani, Matthias Minderer, Georg Heigold, Sylvain Gelly, et al. An image is worth 16x16 words: Transformers for image recognition at scale. *arXiv preprint arXiv:2010.11929*, 2020.

Dheeru Dua and Casey Graff. UCI machine learning repository, 2017. URL `http://archive.ics.uci.edu/ml`.

David K Duvenaud, Dougal Maclaurin, Jorge Iparraguirre, Rafael Bombarell, Timothy Hirzel, Alan Aspuru-Guzik, and Ryan P Adams. Convolutional networks on graphs for learning molecular fingerprints. *Advances in Neural Information Processing Systems*, 28: 2224–2232, 2015.

Harrison Edwards and Amos Storkey. Towards a neural statistician. *arXiv preprint arXiv:1606.02185*, 2016.

Beyza Ermiş, Evrim Acar, and A Taylan Cemgil. Link prediction via generalized coupled tensor factorisation. *arXiv preprint arXiv:1208.6231*, 2012.

Carlos Esteves, Christine Allen-Blanchette, Ameesh Makadia, and Kostas Daniilidis. Learning so (3) equivariant representations with spherical cnns. In *Proceedings of the European Conference on Computer Vision (ECCV)*, pages 52–68, 2018.

Ariel Felner, Richard E Korf, and Sarit Hanan. Additive pattern database heuristics. *Journal of Artificial Intelligence Research*, 22:279–318, 2004.

Yifan Feng, Haoxuan You, Zizhao Zhang, Rongrong Ji, and Yue Gao. Hypergraph neural networks. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 33, pages 3558–3565, 2019.

Marc Finzi, Max Welling, and Andrew Gordon Wilson. A practical method for constructing equivariant multilayer perceptrons for arbitrary matrix groups. In *International Conference on Machine Learning*, pages 3318–3328. PMLR, 2021.

Richard Foote, Gagan Mirchandani, Daniel N. Rockmore, Dennis M. Healy, and Timothy E. Olson. A wreath product group approach to signal and image processing i: Multiresolution analysis. *IEEE Trans. Signal Process.*, 48:102–132, 2000.

Richard Foote, Gagan Mirchandani, and Daniel Rockmore. Two-dimensional wreath product group-based image processing. *Journal of Symbolic Computation*, 37(2):187 – 207, 2004. ISSN 0747-7171. Computer Algebra and Signal Processing.

M. Fox and D. Long. The detection and exploitation of symmetry in planning problems. In *IJCAI*, 1999.

M. Fox and D. Long. Extending the exploitation of symmetries in planning. In *AIPS*, 2002.

Fabian B. Fuchs, Daniel E. Worrall, Volker Fischer, and Max Welling. Se(3)-transformers: 3d roto-translation equivariant attention networks. In *Advances in Neural Information Processing Systems 34 (NeurIPS)*, 2020.

Justin Gilmer, Samuel S Schoenholz, Patrick F Riley, Oriol Vinyals, and George E Dahl. Neural message passing for quantum chemistry. In *International conference on machine learning*, pages 1263–1272. PMLR, 2017.

Linxia Gong, Xiaochuan Feng, Dezhi Ye, Hao Li, Runze Wu, Jianrong Tao, Changjie Fan, and Peng Cui. Optmatch: Optimized matchmaking via modeling the high-order interactions on the arena. In *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, pages 2300–2310, 2020.

Ian Goodfellow, Yoshua Bengio, Aaron Courville, and Yoshua Bengio. *Deep learning*, volume 1. MIT Press, 2016.

Devon Graham, Junhao Wang, and Siamak Ravanbakhsh. Equivariant entity-relationship networks. *arXiv preprint arXiv:1903.09033*, 2019.

Nicholas Guttenberg, Nathaniel Virgo, Olaf Witkowski, Hidetoshi Aoki, and Ryota Kanai. Permutation-equivariant neural networks applied to dynamics prediction. *arXiv preprint arXiv:1612.04530*, 2016.

William L Hamilton, Rex Ying, and Jure Leskovec. Representation learning on graphs: Methods and applications. *arXiv preprint arXiv:1709.05584*, 2017.

P. E. Hart, N. J. Nilsson, and B. Raphael. A formal basis for the heuristic determination of minimum cost paths. *IEEE Transactions on Systems Science and Cybernetics*, 4(2): 100–107, July 1968. ISSN 2168-2887. doi: 10.1109/TSSC.1968.300136.

Jason Hartford, Devon Graham, Kevin Leyton-Brown, and Siamak Ravanbakhsh. Deep models of interactions across sets. In *International Conference on Machine Learning*, pages 1909–1918. PMLR, 2018.

Mikael Henaff, Joan Bruna, and Yann LeCun. Deep convolutional networks on graph-structured data. *arXiv preprint arXiv:1506.05163*, 2015.

Geoffrey E. Hinton, Simon Osindero, and Yee Whye Teh. A fast learning algorithm for deep belief nets. *Neural Computation*, 18:1527–1554, 2006.

Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural computation*, 9(8):1735–1780, 1997.

Weihua Hu, Bowen Liu, Joseph Gomes, Marinka Zitnik, Percy Liang, Vijay Pande, and Jure Leskovec. Strategies for pre-training graph neural networks. In *International Conference on Learning Representations*, 2019.

Jonathan Huang and Carlos Guestrin. Riffled independence for ranked data. In Y. Bengio, D. Schuurmans, J. D. Lafferty, C. K. I. Williams, and A. Culotta, editors, *Advances in Neural Information Processing Systems 22*, pages 799–807. Curran Associates, Inc., 2009.

Jonathan Huang, Carlos Guestrin, and Leonidas J Guibas. Efficient inference for distributions on permutations. In *Advances in neural information processing systems*, pages 697–704, 2008.

Jonathan Huang, Carlos Guestrin, and Leonidas Guibas. Fourier theoretic probabilistic inference over permutations. *Journal of Machine Learning Research*, 10(5), 2009.

Michael J Hutchinson, Charline Le Lan, Sheheryar Zaidi, Emilien Dupont, Yee Whye Teh, and Hyunjik Kim. Lietransformer: Equivariant self-attention for lie groups. In *International Conference on Machine Learning*, pages 4533–4543. PMLR, 2021.

Alexander Irpan. Exploring boosted neural nets for rubik's cube solving. 2016.

Srikanth Jagabathula and Devavrat Shah. Inferring rankings under constrained sensing. In D. Koller, D. Schuurmans, Y. Bengio, and L. Bottou, editors, *Advances in Neural Information Processing Systems 21*, pages 753–760. Curran Associates, Inc., 2009.

James. *The Representation Theory of the Symmetric Group*. Encyclopedia of Mathematics and its Applications. Cambridge University Press, 1984. doi: 10.1017/CBO9781107340732.

Rodolphe Jenatton, Nicolas Le Roux, Antoine Bordes, and Guillaume Obozinski. A latent factor model for highly multi-relational data. In *Advances in Neural Information Processing Systems 25 (NIPS 2012)*, pages 3176–3184, 2012.

David Joyner. *Adventures in Group Theory: Rubik's Cube, Merlin's Machine, and Other Mathematical Toys*. Johns Hopkins University Press, 2 edition, 2008.

Gregor Kasieczka, Tilman Plehn, and Michael Thompson, Jenniferand Russel. Top quark tagging reference dataset, 2019. URL `https://zenodo.org/record/2603256`.

Itay Katzir, Murat Cokol, Bree B Aldridge, and Uri Alon. Prediction of ultra-high-order antibiotic combinations based on pairwise interactions. *PLoS computational biology*, 15 (1):e1006774, 2019.

Seyed Mehran Kazemi and David Poole. Simple embedding for link prediction in knowledge graphs. In *Proceedings of the 32nd International Conference on Neural Information Processing Systems*, pages 4289–4300, 2018.

Philipp W Keller, Shie Mannor, and Doina Precup. Automatic basis function construction for approximate dynamic programming and reinforcement learning. In *Proceedings of the 23rd international conference on Machine learning*, pages 449–456, 2006.

A. Kerber. *Representations of Permutation Groups I*, volume 240 of *Lecture Notes in Mathamatics*. Springer-Verlag Berlin Heidelberg, 1971.

Nicolas Keriven and Gabriel Peyré. Universal invariant and equivariant graph neural networks. *Advances in Neural Information Processing Systems*, 32:7092–7101, 2019.

Jinwoo Kim, Saeyoon Oh, and Seunghoon Hong. Transformers generalize deepsets and can be extended to graphs & hypergraphs. *Advances in Neural Information Processing Systems*, 34, 2021.

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

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

Thomas N Kipf and Max Welling. Semi-supervised classification with graph convolutional networks. *arXiv preprint arXiv:1609.02907*, 2016.

Herbert Kociemba. Two-phase algorithm details. http://kociemba.org/cube.html, n.d. [Online: Accessed: 03/13/2020].

Patrick T Komiske, Eric M Metodiev, and Jesse Thaler. Energy flow polynomials: A complete linear basis for jet substructure. *Journal of High Energy Physics*, 2018(4):1–54, 2018.

Patrick T Komiske, Eric M Metodiev, and Jesse Thaler. Energy flow networks: deep sets for particle jets. *Journal of High Energy Physics*, 2019(1):1–46, 2019.

Imre Risi Kondor. *Group theoretical methods in machine learning*, volume 2. Columbia University New York, 2008.

Risi Kondor. A fourier space algorithm for solving quadratic assignment problems. In *Proceedings of the Twenty-first Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '10, pages 1017–1028, Philadelphia, PA, USA, 2010. Society for Industrial and Applied Mathematics. ISBN 978-0-898716-98-6.

Risi Kondor and Marconi S Barbosa. Ranking with kernels in fourier space. In *COLT*, pages 451–463, 2010.

Risi Kondor and Walter Dempsey. Multiresolution analysis on the symmetric group. In *Advances in Neural Information Processing Systems*, pages 1637–1645, 2012.

Risi Kondor and Shubhendu Trivedi. On the generalization of equivariance and convolution in neural networks to the action of compact groups. In *International Conference on Machine Learning*, pages 2747–2755. PMLR, 2018.

Risi Kondor, Andrew Howard, and Tony Jebara. Multi-object tracking with representations of the symmetric group. In *Artificial Intelligence and Statistics*, pages 211–218, 2007.

Risi Kondor, Zhen Lin, and Shubhendu Trivedi. Clebsch–gordan nets: a fully fourier space spherical convolutional neural network. *Advances in Neural Information Processing Systems*, 31:10117–10126, 2018a.

Risi Kondor, Hy Truong Son, Horace Pan, Brandon Anderson, and Shubhendu Trivedi. Covariant compositional networks for learning graphs. *arXiv preprint arXiv:1801.02144*, 2018b.

George Konidaris, Sarah Osentoski, and Philip Thomas. Value function approximation in reinforcement learning using the fourier basis. In *Proceedings of the Twenty-Fifth AAAI Conference on Artificial Intelligence*, pages 380–385, 2011.

Richard E. Korf. A program that learns to solve Rubik's cube. In David L. Waltz, editor, *Proceedings of the National Conference on Artificial Intelligence, Pittsburgh, PA, USA, August 18-20, 1982*, pages 164–167. AAAI Press, 1982. URL http://www.aaai.org/Library/AAAI/1982/aaai82-039.php.

Richard E Korf. Depth-first iterative-deepening: An optimal admissible tree search. *Artificial intelligence*, 27(1):97–109, 1985.

Richard E Korf. Finding optimal solutions to rubik's cube using pattern databases. In *AAAI/IAAI*, pages 700–705, 1997.

Richard E Korf and Larry A Taylor. Finding optimal solutions to the twenty-four puzzle. In *Proceedings of the national conference on artificial intelligence*, pages 1202–1207, 1996.

R Matthew Kretchmar and Charles W Anderson. Comparison of cmacs and radial basis functions for local function approximators in reinforcement learning. In *Proceedings of International Conference on Neural Networks (ICNN'97)*, volume 2, pages 834–837. IEEE, 1997.

Brenden M Lake, Ruslan Salakhutdinov, and Joshua B Tenenbaum. Human-level concept learning through probabilistic program induction. *Science*, 350(6266):1332–1338, 2015.

Yann LeCun, Bernhard Boser, John S Denker, Donnie Henderson, Richard E Howard, Wayne Hubbard, and Lawrence D Jackel. Backpropagation applied to handwritten zip code recognition. *Neural computation*, 1(4):541–551, 1989.

Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998.

Juho Lee, Yoonho Lee, Jungtaek Kim, Adam Kosiorek, Seungjin Choi, and Yee Whye Teh. Set transformer: A framework for attention-based permutation-invariant neural networks. In *International Conference on Machine Learning*, pages 3744–3753. PMLR, 2019a.

Juho Lee, Yoonho Lee, Jungtaek Kim, Adam Kosiorek, Seungjin Choi, and Yee Whye Teh. Set transformer: A framework for attention-based permutation-invariant neural networks. In Kamalika Chaudhuri and Ruslan Salakhutdinov, editors, *Proceedings of the 36th International Conference on Machine Learning*, volume 97 of *Proceedings of Machine Learning Research*, pages 3744–3753. PMLR, 09–15 Jun 2019b. URL `https://proceedings.mlr.press/v97/lee19d.html`.

Yujia Li, Daniel Tarlow, Marc Brockschmidt, and Richard Zemel. Gated graph sequence neural networks. *arXiv preprint arXiv:1511.05493*, 2015.

Yankai Lin, Zhiyuan Liu, Maosong Sun, Yang Liu, and Xuan Zhu. Learning entity and relation embeddings for knowledge graph completion. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 29, 2015.

Jasper Linmans, Jim Winkens, Bastiaan S Veeling, Taco S Cohen, and Max Welling. Sample efficient semantic segmentation using rotation equivariant convolutional networks. *arXiv preprint arXiv:1807.00583*, 2018.

Qiang Liu, Shu Wu, and Liang Wang. Collaborative prediction for multi-entity interaction with hierarchical representation. In *Proceedings of the 24th ACM International on Conference on Information and Knowledge Management*, pages 613–622, 2015.

Sridhar Mahadevan. Proto-value functions: Developmental reinforcement learning. In *Proceedings of the 22nd international conference on Machine learning*, pages 553–560, 2005.

Sridhar Mahadevan and Mauro Maggioni. Value function approximation with diffusion wavelets and laplacian eigenfunctions. In *Advances in neural information processing systems*, pages 843–850, 2006.

Sridhar Mahadevan and Mauro Maggioni. Proto-value functions: A laplacian framework for learning representation and control in markov decision processes. *Journal of Machine Learning Research*, 8(Oct):2169–2231, 2007.

Horia Mania, Aaditya Ramdas, Martin J Wainwright, Michael I Jordan, Benjamin Recht, et al. On kernel methods for covariates that are rankings. *Electronic Journal of Statistics*, 12(2):2537–2577, 2018.

Haggai Maron, Heli Ben-Hamu, Nadav Shamir, and Yaron Lipman. Invariant and equivariant graph networks. *arXiv preprint arXiv:1812.09902*, 2018.

Nickel Maximilian. *Tensor factorization for relational learning*. PhD thesis, PhD thesis, lmu, 2013. 2 [Google Scholar], 2013.

Stephen McAleer, Forest Agostinelli, Alexander Shmakov, and Pierre Baldi. Solving the rubik's cube with approximate policy iteration. In *International Conference on Learning Representations*, 2018.

Ishai Menache, Shie Mannor, and Nahum Shimkin. Basis function adaptation in temporal difference reinforcement learning. *Annals of Operations Research*, 134(1):215–238, 2005.

Vinicius Mikuni and Florencia Canelli. Abcnet: An attention-based method for particle tagging. *The European Physical Journal Plus*, 135(6):1–11, 2020.

Benjamin Kurt Miller, Mario Geiger, Tess E Smidt, and Frank Noé. Relevance of rotationally equivariant convolutions for predicting molecular properties. *arXiv preprint arXiv:2008.08461*, 2020.

G. Mirchandani, R. Foote, D. Rockmore, D. Healy, and T. Olson. A wreath product group approach to signal and image processing: Part ii - convolution, correlation, and applications, 1999.

Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. Playing atari with deep reinforcement learning. 2013. URL `http://arxiv.org/abs/1312.5602`. NIPS Deep Learning Workshop 2013.

Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A Rusu, Joel Veness, Marc G Bellemare, Alex Graves, Martin Riedmiller, Andreas K Fidjeland, Georg Ostrovski, et al. Human-level control through deep reinforcement learning. *Nature*, 518(7540):529–533, 2015.

Dror Moran, Hodaya Koslowsky, Yoni Kasten, Haggai Maron, Meirav Galun, and Ronen Basri. Deep permutation equivariant structure from motion. *arXiv preprint arXiv:2104.06703*, 2021.

Christopher Morris, Martin Ritzert, Matthias Fey, William L Hamilton, Jan Eric Lenssen, Gaurav Rattan, and Martin Grohe. Weisfeiler and leman go neural: Higher-order graph neural networks. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 33, pages 4602–4609, 2019.

Christopher Morris, Gaurav Rattan, Sandra Kiefer, and Siamak Ravanbakhsh. Speqnets: Sparsity-aware permutation-equivariant graph networks. *arXiv preprint arXiv:2203.13913*, 2022.

Deepak Nathani, Jatin Chauhan, Charu Sharma, and Manohar Kaul. Learning attention-based embeddings for relation prediction in knowledge graphs. In *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics*, pages 4710–4723, 2019.

Maximilian Nickel and Volker Tresp. Logistic tensor factorization for multi-relational data. *arXiv preprint arXiv:1306.2084*, 2013.

Maximilian Nickel, Volker Tresp, and Hans-Peter Kriegel. A three-way model for collective learning on multi-relational data. In *Proceedings of the 28th International Conference on International Conference on Machine Learning*, pages 809–816, 2011.

Maximilian Nickel, Kevin Murphy, Volker Tresp, and Evgeniy Gabrilovich. A review of relational machine learning for knowledge graphs. *Proceedings of the IEEE*, 104(1):11–33, 2015.

Liang Pang, Jun Xu, Qingyao Ai, Yanyan Lan, Xueqi Cheng, and Jirong Wen. Setrank: Learning a permutation-invariant ranking model for information retrieval. In *Proceedings of the 43rd International ACM SIGIR Conference on Research and Development in Information Retrieval*, pages 499–508, 2020.

Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. Pytorch: An imperative style, high-performance deep learning library. *Advances in neural information processing systems*, 32:8026–8037, 2019.

Przemysław Pobrotyn, Tomasz Bartczak, Mikołaj Synowiec, Radosław Białobrzeski, and Jarosław Bojar. Context-aware learning to rank with self-attention. *arXiv preprint arXiv:2005.10084*, 2020.

Nir Pochter, Aviv Zohar, and Jeffrey S Rosenschein. Exploiting problem symmetries in state-based planners. In *Twenty-Fifth AAAI Conference on Artificial Intelligence*, 2011.

Ira Pohl. Heuristic search viewed as path finding in a graph. *Artificial Intelligence*, 1(3):193 – 204, 1970. ISSN 0004-3702. doi: https://doi.org/10.1016/0004-3702(70)90007-X. URL http://www.sciencedirect.com/science/article/pii/000437027090007X.

Martin L. Puterman. *Markov Decision Processes: Discrete Stochastic Dynamic Programming*. John Wiley and Sons, Inc., USA, 1st edition, 1994. ISBN 0471619779.

Charles R Qi, Hao Su, Kaichun Mo, and Leonidas 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*, pages 652–660, 2017.

Huilin Qu and Loukas Gouskos. Jet tagging via particle clouds. *Physical Review D*, 101(5): 056019, 2020.

Siamak Ravanbakhsh, Jeff Schneider, and Barnabás Póczos. Equivariance through parameter-sharing. In Doina Precup and Yee Whye Teh, editors, *Proceedings of the 34th International Conference on Machine Learning*, volume 70 of *Proceedings of Machine Learning Research*, pages 2892–2901. PMLR, 06–11 Aug 2017. URL https://proceedings.mlr.press/v70/ravanbakhsh17a.html.

Hongyu Ren, Hanjun Dai, Zihang Dai, Mengjiao Yang, Jure Leskovec, Dale Schuurmans, and Bo Dai. Combiner: Full attention transformer with sparse computation cost. *Advances in Neural Information Processing Systems*, 34, 2021.

Dan Rockmore, Peter Kostelec, Wim Hordijk, and Peter F. Stadler. Fast Fourier transform for fitness landscapes. *Applied and Computational Harmonic Analysis*, 12(1):57 – 76, 2002. ISSN 1063-5203. doi: https://doi.org/10.1006/acha.2001.0346. URL http://www.sciencedirect.com/science/article/pii/S106352030190346X.

Daniel N Rockmore. Fast Fourier transforms for wreath products. *Applied and Computational Harmonic Analysis*, 2(3):279–292, 1995.

David E Rumelhart, Geoffrey E Hinton, and Ronald J Williams. Learning internal representations by error propagation. Technical report, California Univ San Diego La Jolla Inst for Cognitive Science, 1985.

Bruce Sagan. *The symmetric group: representations, combinatorial algorithms, and symmetric functions*, volume 203. Springer Science & Business Media, 2001.

Bruce E Sagan. *The symmetric group: representations, combinatorial algorithms, and symmetric functions*, volume 203. Springer Science & Business Media, 2013.

Victor Garcia Satorras, Emiel Hoogeboom, and Max Welling. E (n) equivariant graph neural networks. In *International Conference on Machine Learning*, pages 9323–9332. PMLR, 2021.

Franco Scarselli, Marco Gori, Ah Chung Tsoi, Markus Hagenbuchner, and Gabriele Monfardini. The graph neural network model. *IEEE transactions on neural networks*, 20(1): 61–80, 2008.

Michael Schlichtkrull, Thomas N Kipf, Peter Bloem, Rianne Van Den Berg, Ivan Titov, and Max Welling. Modeling relational data with graph convolutional networks. In *European semantic web conference*, pages 593–607. Springer, 2018.

John Schulman, Sergey Levine, Pieter Abbeel, Michael Jordan, and Philipp Moritz. Trust region policy optimization. In *International conference on machine learning*, pages 1889–1897, 2015.

John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347*, 2017.

Nimrod Segol and Yaron Lipman. On universal equivariant set networks. In *International Conference on Learning Representations*, 2019.

Ákos Seress. *Permutation Group Algorithms*. Cambridge Tracts in Mathematics. Cambridge University Press, 2003. doi: 10.1017/CBO9780511546549.

Jean-Pierre Serre. *Linear representations of finite groups*, volume 42. Springer, 1977a.

Jean-Pierre Serre. *Linear representations of finite groups*, volume 42. Springer, 1977b.

Chao Shang, Yun Tang, Jing Huang, Jinbo Bi, Xiaodong He, and Bowen Zhou. End-to-end structure-aware convolutional networks for knowledge base completion. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 33, pages 3060–3067, 2019.

Claude E. Shannon. A mathematical theory of communication. *Bell System Technical Journal*, 27(3):379–423, 1948.

Yifeng Shi, Junier Oliva, and Marc Niethammer. Deep message passing on sets. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 34, pages 5750–5757, 2020.

Alexander Shleyfman, Michael Katz, Malte Helmert, Silvan Sievers, and Martin Wehrle. Heuristics and symmetries in classical planning. In *Proceedings of the Twenty-Ninth AAAI Conference on Artificial Intelligence*, AAAI'15, page 3371–3377. AAAI Press, 2015. ISBN 0262511290.

David Silver, Aja Huang, Chris J Maddison, Arthur Guez, Laurent Sifre, George Van Den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, et al. Mastering the game of go with deep neural networks and tree search. *nature*, 529(7587):484–489, 2016.

Richard Socher, Danqi Chen, Christopher D Manning, and Andrew Ng. Reasoning with neural tensor networks for knowledge base completion. In *Advances in neural information processing systems*, pages 926–934. Citeseer, 2013.

Aravind Srinivas, Michael Laskin, and Pieter Abbeel. Curl: Contrastive unsupervised representations for reinforcement learning. *arXiv preprint arXiv:2004.04136*, 2020.

Sainbayar Sukhbaatar, Rob Fergus, et al. Learning multiagent communication with backpropagation. *Advances in neural information processing systems*, 29, 2016.

Zhiqing Sun, Zhi-Hong Deng, Jian-Yun Nie, and Jian Tang. Rotate: Knowledge graph embedding by relational rotation in complex space. In *International Conference on Learning Representations*, 2018.

Ilya Sutskever, Ruslan Salakhutdinov, and Joshua B Tenenbaum. Modelling relational data using bayesian clustered tensor factorization. In *Proceedings of the 22nd International Conference on Neural Information Processing Systems*, pages 1821–1828, 2009.

Richard S. Sutton. Learning to predict by the methods of temporal differences. *Mach. Learn.*, 3(1):9–44, August 1988. ISSN 0885-6125. doi: 10.1023/A:1022633531479. URL `https://doi.org/10.1023/A:1022633531479`.

Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction*. The MIT Press, second edition, 2018.

Richard S Sutton, David A McAllester, Satinder P Singh, and Yishay Mansour. Policy gradient methods for reinforcement learning with function approximation. In *Advances in neural information processing systems*, pages 1057–1063, 2000.

Jerry Swan. Harmonic analysis and resynthesis of sliding-tile puzzle heuristics. In *2017 IEEE Congress on Evolutionary Computation (CEC)*, pages 516–524. IEEE, 2017.

Elif Tekin, Cynthia White, Tina Manzhu Kang, Nina Singh, Mauricio Cruz-Loya, Robert Damoiseaux, Van M Savage, and Pamela J Yeh. Prevalence and patterns of higher-order drug interactions in escherichia coli. *NPJ systems biology and applications*, 4(1):1–10, 2018.

Audrey Terras. *Fourier analysis on finite groups and applications*. London Mathematical Society Student Texts. Cambridge University Press, 1999.

Erik Henning Thiede, Truong Son Hy, and Risi Kondor. The general theory of permutation equivarant neural networks and higher order graph variational encoders. *arXiv preprint arXiv:2004.03990*, 2020.

Théo Trouillon, Johannes Welbl, Sebastian Riedel, Éric Gaussier, and Guillaume Bouchard. Complex embeddings for simple link prediction. In *International Conference on Machine Learning*, pages 2071–2080. PMLR, 2016.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. *Advances in neural information processing systems*, 30, 2017.

Petar Veličković, Guillem Cucurull, Arantxa Casanova, Adriana Romero, Pietro Liò, and Yoshua Bengio. Graph Attention Networks. *International Conference on Learning Representations*, 2018. URL `https://openreview.net/forum?id=rJXMpikCZ`.

Clement Vignac, Andreas Loukas, and Pascal Frossard. Building powerful and equivariant graph neural networks with structural message-passing. *Advances in Neural Information Processing Systems*, 33:14143–14155, 2020.

Oriol Vinyals, Samy Bengio, and Manjunath Kudlur. Order matters: Sequence to sequence for sets. *arXiv preprint arXiv:1511.06391*, 2015.

Edward Wagstaff, Fabian Fuchs, Martin Engelcke, Ingmar Posner, and Michael A Osborne. On the limitations of representing functions on sets. In *International Conference on Machine Learning*, pages 6487–6494. PMLR, 2019.

Edward Wagstaff, Fabian B Fuchs, Martin Engelcke, Michael A Osborne, and Ingmar Posner. Universal approximation of functions on sets. *arXiv preprint arXiv:2107.01959*, 2021.

Zhen Wang, Jianwen Zhang, Jianlin Feng, and Zheng Chen. Knowledge graph embedding by translating on hyperplanes. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 28, 2014.

Christopher J. C. H. Watkins and Peter Dayan. Q-learning. In *Machine Learning*, pages 279–292, 1992a.

Christopher J. C. H. Watkins and Peter Dayan. Q-learning. In *Machine Learning*, pages 279–292, 1992b.

Maurice Weiler, Mario Geiger, Max Welling, Wouter Boomsma, and Taco Cohen. 3d steerable cnns: learning rotationally equivariant features in volumetric data. In *Proceedings of the 32nd International Conference on Neural Information Processing Systems*, pages 10402–10413, 2018.

Jeffrey Wood and John Shawe-Taylor. Representation theory and invariant neural networks. *Discrete applied mathematics*, 69(1-2):33–60, 1996.

Daniel Worrall and Gabriel Brostow. Cubenet: Equivariance to 3d rotation and translation. In *Proceedings of the European Conference on Computer Vision (ECCV)*, pages 567–584, 2018.

Saining Xie, Ross Girshick, Piotr Dollár, Zhuowen Tu, and Kaiming He. Aggregated residual transformations for deep neural networks. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 1492–1500, 2017.

Yunyang Xiong, Zhanpeng Zeng, Rudrasis Chakraborty, Mingxing Tan, Glenn Fung, Yin Li, and Vikas Singh. Nyströmformer: A nyström-based algorithm for approximating self-attention. In *Proceedings of the... AAAI Conference on Artificial Intelligence. AAAI Conference on Artificial Intelligence*, volume 35, page 14138. NIH Public Access, 2021.

Keyulu Xu, Weihua Hu, Jure Leskovec, and Stefanie Jegelka. How powerful are graph neural networks? In *International Conference on Learning Representations*, 2018.

Bishan Yang, Scott Wen-tau Yih, Xiaodong He, Jianfeng Gao, and Li Deng. Embedding entities and relations for learning and inference in knowledge bases. In *Proceedings of the International Conference on Learning Representations (ICLR) 2015*, May 2015.

Dmitry Yarotsky. Universal approximations of invariant maps by neural networks. *Constructive Approximation*, 55(1):407–474, 2022.

Kenan Yılmaz, Ali Cemgil, and Umut Simsekli. Generalised coupled tensor factorisation. *Advances in neural information processing systems*, 24:2151–2159, 2011.

Manzil Zaheer, Satwik Kottur, Siamak Ravanbakhsh, Barnabas Poczos, Russ R Salakhutdinov, and Alexander J Smola. Deep sets. *Advances in Neural Information Processing Systems*, 30, 2017.

Anat Zimmer, Itay Katzir, Erez Dekel, Avraham E Mayo, and Uri Alon. Prediction of multidimensional drug dose responses based on measurements of drug pairs. *Proceedings of the National Academy of Sciences*, 113(37):10442–10447, 2016.