Permutation Equivariant Layers for Higher Order Interactions

Horace Pan
University of Chicago
Department of Computer Science
hopan@uchicago.edu

Risi Kondor
University of Chicago
Department of Computer Science, Statistics
University of Chicago
risi@uchicago.edu

Abstract

Recent work on permutation equivariant neural networks has mostly focused on the first order case (sets) and second order case (graphs). We describe the machinery for generalizing permutation equivariance to arbitrary k-ary interactions between entities for any value of k. We demonstrate the effectiveness of higher order permutation equivariant models on several real world applications and find that our results compare favorably to existing permutation invariant/equivariant baselines.

1 INTRODUCTION

Various machine learning problems involve modeling the interactions between individual objects from a given set, or between multiple sets. In graph learning problems, we have vertices representing entities and the (hyper-)edges encoding information about how vertices are related. Learning on unordered sets of elements such as point cloud data is another common scenario. Finally, in relation learning problems we are explicitly presented with relationships between entities and the task is to infer other relationships.

A fundamental mathematical requirement in all these scenarios is that the learned relationships must not be dependent on the (arbitrary) way that we choose to number the entities, i.e., that the learning algorithm should be *equivariant* to permutations. In this paper we consider the problem of constructing the most general type of neural network layers that can capture k'th order relationships in a permutation equivariant way.

Proceedings of the 25$^{\text{th}}$ International Conference on Artificial Intelligence and Statistics (AISTATS) 2022, Valencia, Spain. PMLR: Volume 151.

The activations in such networks are k'th order *permutable* tensors, which we formally define in Section 2. We describe a general framework for enumerating and efficiently computing all possible permutation equivariant linear maps between such tensors and derive the computational complexity of all of the operations involved.

1.1 Previous/Related Work

Equivariance Equivariance is a crucial design principle for constructing neural networks for learning on structured data or data that exhibits symmetries. Equivariant architectures include classical convolutional neural networks (CNNs), which leverage the translational symmetry in image recognition (LeCun et al., 1998), spherical CNNs (Cohen et al., 2018; Weiler et al., 2018; Kondor et al., 2018a; Esteves et al., 2018), graph neural networks (Scarselli et al., 2008; Kipf and Welling, 2016; Battaglia et al., 2018), and permutation invariant neural network architectures (Qi et al., 2017; Zaheer et al., 2017; Maron et al., 2018; Vaswani et al., 2017) to name a few.

Graph Learning Modeling data with relationship between entities is most commonly done in the graph learning setting where relationships can be encoded as edges in the graph. Early work on graph networks focused on learning functions on the graph using the graph Fourier basis (Bruna et al., 2013; Henaff et al., 2015), which is invariant to permutations of the vertices of the graph. Subsequent graph neural network architectures (Duvenaud et al., 2015; Li et al., 2015; Kipf and Welling, 2016; Gilmer et al., 2017) perform message passing and message aggregation at each node of the graph in a permutation invariant manner. To deal with higher order interactions between nodes, several works extend graph neural network ideas to the hypergraph setting (Feng et al., 2019; Dong et al., 2020; Bai et al., 2021) by applying the same message passing procedure on the hypernodes and hyperedges of the hypergraph.

Permutation Invariant/Equivariant Networks Architectures such as PointNet (Qi et al., 2017) for learning on point clouds and Deep Sets (Zaheer et al., 2017) for learning on sets of structured data were among the first to recognize that when dealing with collections of objects, it is important to have the neural network be invariant to permuting the ordering of the objects. Deep Sets showed that permutation invariant and equivariant architectures must abide by a very simple form involving a symmetric pooling operation over the inputs to enforce permutation invariance. Kondor et al. (2018b) prove that elementary tensor operations such as as tensor products, elementwise operations, and tensor contractions are permutation equivariant operations and show that permutation equivariant neural networks can be constructed by interleaving the linear tensor operations with the pointwise nonlinearities and tensor products. Thiede et al. (2020) derive a form for second order permutation equivariant layers and apply them in graph variational autoencoders.

Given how popular they have become over the past four years, we would be remiss not to mention attention based models. Attention layers have been used with great success in neural networks for learning tasks on sequences (Bahdanau et al., 2015; Vaswani et al., 2017), graphs (Veličković et al., 2018), sets (Lee et al., 2019), 3D objects (Fuchs et al., 2020) and images Dosovitskiy et al. (2020). Though they were originally used to model text/language data and not necessarily to address permutation symmetry in data, attention layers do in fact provide a permutation equivariant featurization for the entities being "attended" to.

Hartford et al. (2018) describe a permutation equivariant architecture for modeling interactions between elements across different sets (e.g. user-movie ratings). which come in the form interaction matrices $\boldsymbol{X} \in \mathbb{R}^{N\times M}$. Permutations can act on this matrix along the users dimension by $\mathbb{S}_N$ or along the movies dimension by $\mathbb{S}_M$. Any permutation equivariant layer applied to $\boldsymbol{X}$ must satisfy a certain parameter sharing scheme. They further show that in the case where the two sets have equal size, there are exactly 4 free parameters in the weight matrix. They also provide a few conditions on how the weight matrix underlying the equivariant layer must have a specific parameter sharing scheme. Graham et al. (2019) describes a similar parameter sharing scheme for permutation equivariant architectures applied to entity-relationship networks. Parameter sharing for equivariance with respect to discrete groups is also discussed in Ravanbakhsh et al. (2017).

Maron et al. (2018) bears the most similarity to our present work. Following some of the aforementioned earlier works on permutation equivariance, they provided a derivation of the number of free parameters allowed in permutation equivariant layers between permutable tensors of arbitrary order and give a full characterization of the parameter sharing scheme. They show that the space of permutation equivariant linear layers from $\mathbb{R}^{n_1^k} \to \mathbb{R}^{n_2^k}$ has dimension $B(k_1+k_2)$, where $B(n)$ denotes the nth Bell number which corresponds to the number of distinct parameters that are possible over the equivalence classes of entries of the weight matrices in the equivariant layer. While their work describes the number of free parameters in the weight matrices and their parameter sharing scheme, they do not give details on how to actually construct permutation equivariant layers except when $k_1 = k_2 = 2$.

Paper organization Section 2 introduces some of the notation that will be used throughout the paper. Sections 3 and 4 describe how to derive the necessary equivariant operations. Section 5 discusses how to organize the computation in the proposed layers. Finally, section 6 presents the results of permutation equivariant architectures that feature our 2nd and 3rd order equivariant layers on two set learning tasks.

2 PRELIMINARIES

Definition 2.1 (Symmetric group)**.** The symmetric group of degree n, denoted $\mathbb{S}_n$, is the set of all bijections from $\{1, 2, \dots, n\}$ to $\{1, 2, \dots, n\}$.

Definition 2.2 (Permutable k'th order tensors)**.** A k'th order permutable tensor over a set of entities $\{e_1, \dots, e_n\}$ is a multidimensional array $\boldsymbol{T}_k \in \mathbb{R}^{n^k \times d}$. The last dimension of $\boldsymbol{T}_k$ we call the *feature dimension*.

For simplicity, in much of our general discussion of equivariance we just set $d = 1$ and drop the feature index. The feature index is then reintroduced in the supplement, where we prove that mixing feature channels commutes with the other operations in the network.

The fact that each of the first k indices of $\boldsymbol{T}_k$ are associated with $\{e_1, \dots, e_n\}$ implies that under a permutation $\sigma \in \mathbb{S}_n$ of these entities, $(e_1, \dots, e_n) \mapsto (e_{\sigma(1)}, \dots, e_{\sigma(n)})$, the tensor $\boldsymbol{T}_k$ transforms as

$$\boldsymbol{T}_k \mapsto \sigma \cdot \boldsymbol{T}_k \tag{1}$$

$$[\sigma \cdot \boldsymbol{T}_k]_{i_1,\dots i_k} = [\boldsymbol{T}_k]_{\sigma^{-1}(i_1),\dots,\sigma^{-1}(i_k)}. \tag{2}$$

Formally, we say that the symmetric group $\mathbb{S}_n$ *acts* on permutable tensors by the action (1).

The type of neural architectures that we are concerned with in this paper involve multiple permutable tensors (potentially of different orders) related to each other

by *learnable* mappings. The constraint on such mappings is that if their inputs change according to (1), their outputs must also change according to a similar action. This is the constraint that is captured by the concept of *permutation equivariance*.

Definition 2.3 (Permutation Equivariance)**.** A mapping $\phi : \mathbb{R}^{n^{k_1} \times d_1} \to \mathbb{R}^{n^{k_2} \times d_2}$ between k_1'th and k_2'th order permutable tensors is is said to be **permutation equivariant** if

$$\phi(\sigma \cdot \boldsymbol{T}_{k_1}) = \sigma \cdot \phi(\boldsymbol{T}_{k_1}),$$

for all permutations $\sigma \in \mathbb{S}_n$, and all $\boldsymbol{T}_{k_1}$.

Element-wise nonlinearities are permutation equivariant since the operation commutes with the permutation action. Following modern neural network design principles, we construct a permutation invariant neural network by interleaving linear permutation equivariant layers with permutation equivariant nonlinearities (e.g.: ReLU's) and eventually applying a permutation invariant pooling operation before the output layer. Tensor products and outer products of permutable tensors are two other examples of simple nonlinear permutation equivariant operations that one can use in constructing these networks. Therefore in much of this paper, we will focus on *linear* permutation equivariant layers, in particular because it is these layers which will involve learnable parameters.

2.1 Examples of permutable tensors

First order permutable tensors $\boldsymbol{T}_1 \in \mathbb{R}^{n \times d}$ capture individual properties of the entities $e_1, e_2, \dots, e_n$. The ith row of T_1 corresponds to a feature vector for entity e_i. This form of permutation equivariance was first explored in (Zaheer et al., 2017) and has since found numerous applications such as jet tagging in particle physics (Komiske et al., 2019; Dolan and Ore, 2021), and structure from motion in computer vision (Moran et al., 2021).

Second order permutable tensors appear in any setting where a neural network learns from relations between pairs of entities or learns in a way that induces relations between pairs of entities. Such second order relationships can be either symmetric or antisymmetric. An example of a symmetric relationship might be "words e_i and e_j co-occur in a document" whereas an example of an antisymmetric relationship is "player e_i beat player e_j in a chess tournament."

The prototypical example of second order relationships is graphs, where $\{e_1, \dots, e_n\}$ is identified with the vertices and the relationship is simply that e_i and e_j are adjacent to each other. By definition, a graph is a topological object and whatever ordering we impose on its vertices to feed it in our neural network is wholly arbitrary. The output of the neural network must not depend on this arbitrary ordering, so, as discussed by several authors (Keriven and Peyré, 2019; Maron et al., 2018), permutation equivariance is a fundamental requirement for graph learning.

Higher order relationships appear when we consider interactions between $k \geq 3$ entities. An example that appears in our experiments is learning the interactions between multiple drugs (Tekin et al., 2018). Other scenarios might include, for example, predicting the performance of a team based on the players (Gong et al., 2020) or learning to rank based on interactions between items in a list (Pobrotyn et al., 2020).

3 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 by reasoning about what symmetric linear operations are possible.

3.1 Case I: $T_1 \mapsto T_1$ layer

A first order permutable tensor is just a vector whose elements are permuted by σ the natural way:

$$[\sigma \cdot \boldsymbol{T}_1]_i = [\boldsymbol{T}_1]_{\sigma^{-1}(i)}.$$

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

1. The identity map (multiplied by a scalar):
$$\boldsymbol{Y}_i \leftarrow \lambda_1 \boldsymbol{X}_i$$
2. The averaging map (multiplied by a scalar):
$$\boldsymbol{Y}_i \leftarrow \lambda_2 \sum_j \boldsymbol{X}_j$$

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

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

giving us only two learnable parameters λ_1 and λ_2.

3.2 Case II: $T_1 \to T_2$ layer

Now consider a linear permutation equivariant function that maps a first order permutable tensor $\boldsymbol{X} \in \mathbb{R}^n$ to a second order tensor $\boldsymbol{Y} \in \mathbb{R}^{n \times n}$. Let us consider

how we might construct entry $\boldsymbol{Y}_{ij}$ in a symmetric fashion. We have two types of entries to be updated: the diagonal and off-diagonal entries. The off-diagonal entries $\boldsymbol{Y}_{ij}$ can be set to a linear combination of $\boldsymbol{X}_i, \boldsymbol{X}_j$, and the sum of all the $\boldsymbol{X}_k$'s. The diagonal entries $\boldsymbol{Y}_{ii}$ by themselves form a first order permutable tensor, so following Case I, they can be set to a linear combination of $\boldsymbol{X}_i$ and $\sum_k \boldsymbol{X}_k$:

$$\boldsymbol{Y}_{ij} \leftarrow \lambda_1 \boldsymbol{X}_i + \lambda_2 \boldsymbol{X}_j + \lambda_3 (\sum_k \boldsymbol{X}_k)$$
$$\boldsymbol{Y}_{ii} \leftarrow \lambda_4 \boldsymbol{X}_i + \lambda_5 (\sum_k \boldsymbol{X}_k)$$

Thus, $\boldsymbol{Y}$ is a linear combination of five second order permutable tensors:

- a tensor with $\boldsymbol{X}$ broadcast over the rows of the output tensor
- a tensor with $\boldsymbol{X}$ broadcast over the columns of the output tensor
- a tensor with $\boldsymbol{X}$ embedded in the diagonal
- a tensor with the global sum of $\boldsymbol{X}$ tiled over all entries
- a tensor with the global sum of of $\boldsymbol{X}$ tiled over the diagonal

3.3 Case III: $T_2 \to T_1$ layer

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

$$\boldsymbol{Y}_i \leftarrow \lambda_1 \boldsymbol{X}_{ii} + \lambda_2 \sum_j \boldsymbol{X}_{ij} + \lambda_3 \sum_j \boldsymbol{X}_{ji} + \lambda_4 \sum_{ij} \boldsymbol{X}_{ij} + \lambda_5 \sum_i \boldsymbol{X}_{ii}$$

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

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

3.4 Case IV: T_2 to T_2 Equivariant Layer

The expected number of parameters for an equivariant layer mapping from a 2nd order to 2nd order is $B(2+2) = 15$. Figuring out all the symmetric ways of aggregating terms in $\boldsymbol{X}$ that may affect $\boldsymbol{Y}_{ij}$ starts to get a bit cumbersome. In the supplement, we give a full enumeration of all the ways to construct the 2nd order tensors that go into the resulting output tensor which Maron et al. (2018) also describes. While it is still somewhat manageable to enumerate all possible operations in the $\boldsymbol{T}_2 \to \boldsymbol{T}_2$ equivariant layer, it quickly becomes untenable to try to figure out the necessary operations and tensors involved in layers beyond 2nd order. We would like to have a systematic way enumerating the linear combination of tensors that must be a part of the output of the layer.

4 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 1 (Maron et al. (2018))**.** A matrix $\boldsymbol{M} \in \mathbb{R}^{n^{k_2} \times n^{k_1}}$ is a permutation equivariant linear map from $\mathbb{R}^{n^{k_1}} \to \mathbb{R}^{n^{k_2}}$ if and only if $\boldsymbol{M}$ is invariant to its permutation action. If we view $\boldsymbol{M}$ as a (k_1+k_2)-th multidimensional array. The permutation action on $\boldsymbol{M}$ is given by:

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

For any two indices written as a (k_1+k_2)-tuple: $I_1 = (i_1, \dots, i_{k_1+k_2})$ and $I_2 = (j_1, \dots, j_{k_1+k_2})$ where each of the indices of I_1 and I_2 are in $\{1, \dots, n\}$, if $\sigma \cdot I_1 = I_2$, then $\boldsymbol{M}_{I_1} = \boldsymbol{M}_{I_2}$. In other words, the values of $\boldsymbol{M}$ are the same along indices with the same partition pattern. Therefore, $\boldsymbol{M}$ 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 $\boldsymbol{M}$, it is necessary to inspect its indices corresponding to the same partition pattern in $\boldsymbol{M}$ for every partition $\mathcal{P}$ of $\{1, 2, \dots, k_1+k_2\}$.

Definition 4.1 (Bell Number)**.** The n'th Bell number, denoted $B(n)$, counts the number of different ways to partition n into a non-empty subsets.

Definition 4.2 (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, unlabeled subsets.

The following notation is used to index elements of a kth order tensor according to partitions of k. Given a kth order tensor $\boldsymbol{M} \in \mathbb{R}^{n^k}$, and a partition $\mathcal{P}$ of $\{1, \dots, k\}$, we use the notation $\boldsymbol{M}_{\mathcal{P}}$ to denote the set of indices of $\boldsymbol{M}$ 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 1. An adjacency matrix $\boldsymbol{A}$ 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 $\boldsymbol{A}$ corresponding to partition $\big\{\{1, 2\}\big\}$ are the diagonal entries: $\boldsymbol{A}_{\mathcal{P}_1} = \{\boldsymbol{A}_{ij} \mid i = j\}$. The entries of $\boldsymbol{A}$ that correspond to partition $\big\{\{1\}, \{2\}\big\}$ are the off-diagonal entries: $\boldsymbol{A}_{\mathcal{P}_2} = \{\boldsymbol{A}_{ij} \mid i \neq j\}$.

Example 2. Let $\boldsymbol{M} \in \mathbb{R}^{n \times n \times n}$ be a third order permutable tensor. There are five partitions of $\{1, 2, 3\}$, which are: $\big\{\{1, 2, 3\}\big\}, \big\{\{1, 2\}, \{3\}\big\}, \big\{\{1, 3\}, \{2\}\big\}, \big\{\{2, 3\}, \{1\}\big\}$, and $\big\{\{1\}, \{2\}, \{3\}\big\}$. Then we have five equivalence classes of entries of $\boldsymbol{M}$, each corresponding to one of the partitions.

The first partition: $\big\{\{1, 2, 3\}\big\}$ describes the diagonal entries of $\boldsymbol{M}$: $\{\boldsymbol{M}_{ijk} \mid i = j = k\}$. The 2nd, 3rd, and 4th partitions describe entries of $\boldsymbol{M}$ with two indices with the same value and the remaining index being different from the other two. Finally, the last partition $\big\{\{1\}, \{2\}, \{3\}\big\}$ corresponds to the entries of $\boldsymbol{M}$ where all the indices are different: $\{\boldsymbol{M}_{ijk} \mid i \neq j \neq k\}$

We now introduce some convenient notation for describing linear mappings between k_1 and k_2th order tensors. If we have $\boldsymbol{T}_1$ a k_1th order tensor and $\boldsymbol{T}_2$ a k_2th order tensor, such that the entries of $\boldsymbol{T}_2$ are a linear function of the entries of $\boldsymbol{T}_1$. By linearity, we can express indices of $\boldsymbol{T}_2$ as:

$$[\boldsymbol{T}_2]_I = \sum_{J \in [n]^{k_1}} \boldsymbol{M}_{I,J} [\boldsymbol{T}_1]_J$$

where $I \in [n]^{k_2}$ is a k_2-tuple of the integers $1, \dots, n$ and $\boldsymbol{M} \in \mathbb{R}^{k_1 + k_2}$ contains the coefficients underlying the linear function that maps $\boldsymbol{T}_1$ to $\boldsymbol{T}_2$. $M_{I,J}$ denotes the index of $\boldsymbol{M}$ associated with the $k_1 + k_2$ tuple of the concatenation of tuples I and J. We also say that $\boldsymbol{T}_2 = \boldsymbol{M} \cdot \boldsymbol{T}_1$

To better understand the implications of $\boldsymbol{M}$ being constant on each equivalence class of indices, let $\boldsymbol{T}_1$ and $\boldsymbol{T}_2$ be 3rd order tensors where $\boldsymbol{T}_2 = \phi(\boldsymbol{T}_1)$ for some linear permutation equivariant mapping $\phi : \mathbb{R}^{n^{k_1}} \to \mathbb{R}^{n_{k_1}}$. Denote $\boldsymbol{M} \in \mathbb{R}^{n^{k_2} \times n^{k_1}}$ as the matrix such that $\boldsymbol{T}_2 = \boldsymbol{M} \cdot \boldsymbol{T}_1$. We will also use the notation $\boldsymbol{M}_{[\mathcal{P}]} \in \mathbb{R}$ to denote the shared scalar value in the entries of $\boldsymbol{M}$ 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:

$$[\boldsymbol{T}_2]_{i_1, i_2, i_2} \leftarrow \sum_{i_3} \boldsymbol{M}_{i_1, i_2, i_2 | i_2, i_3, i_3} [\boldsymbol{T}_1]_{i_2, i_3, i_3} \tag{3}$$

$$[\boldsymbol{T}_2]_{i_1, i_2, i_2} \leftarrow \boldsymbol{M}_{[\mathcal{P}]} \cdot \sum_{i_3} [\boldsymbol{T}_1]_{i_2, i_3, i_3} \tag{4}$$

Recall, that the elements of the weight matrix underlying a permutation equivariant layer must be constant on indices with the same partition pattern, hence $\boldsymbol{M}_{i_1, i_2, i_2 | i_2, i_3, i_3} = \boldsymbol{M}_{[\mathcal{P}]}$ for all distinct i_1, i_2, i_3. So we can pull the constant $\boldsymbol{M}_{[\mathcal{P}]}$ out of the summation in Eqn. (4).

The full linear map can be computed by evaluating the the same update (equation (3)) for every partition $\mathcal{P}$. The advantage of breaking down our computation of $\boldsymbol{T}_2$ in this manner is that we can systematically isolate the operations that are applied only on $\boldsymbol{T}_1$ and operations that need to be applied to $\boldsymbol{T}_2$. In our example above, the operation applied to $\boldsymbol{T}_1$ amounted to a summation over two of its indices. which can be efficiently computed by numerical libraries.

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**.

5 ORGANIZING THE COMPUTATION

The summation appearing in Eqn. (3) 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 $\boldsymbol{T}_2$. There are three key quantities for organizing this computation: (a) n_s, the number of indices on the right of the dividing line that are associated with summation variables, (b) n_b, the number of indices on the left which are associated with the broadcast variables, (c) κ, the number of transfer variables.

5.0.1 Sums

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) 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 3. 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 $k_1 = 3$. In the supplement we show exactly what these 15 tensors are.

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.

5.0.2 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]

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

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

The transfer variables in Equation (3) tell us where to send the result of our intermediate summation operations.

Example 4. 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: $[\boldsymbol{T}_2]_{i_1,i_2,i_3} \leftarrow \boldsymbol{M}_{[\mathcal{P}]}[\boldsymbol{T}_1]_{i_1,i_2,i_3}$. In matrix form, it is: $\boldsymbol{T}_2 \leftarrow \boldsymbol{M}_{[\mathcal{P}]}\boldsymbol{T}_1$.

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:

$$[\boldsymbol{T}_2]_{i_1i_2i_3} \leftarrow \lambda_1[\boldsymbol{T}_1]_{i_1,i_2,i_3}, \quad [\boldsymbol{T}_2]_{i_1i_3i_2} \leftarrow \lambda_2[\boldsymbol{T}_1]_{i_1,i_2,i_3}$$
$$[\boldsymbol{T}_2]_{i_2i_1i_3} \leftarrow \lambda_3[\boldsymbol{T}_1]_{i_1,i_2,i_3}, \quad [\boldsymbol{T}_2]_{i_2i_3i_1} \leftarrow \lambda_4[\boldsymbol{T}_1]_{i_1,i_2,i_3}$$
$$[\boldsymbol{T}_2]_{i_3i_1i_2} \leftarrow \lambda_5[\boldsymbol{T}_1]_{i_1,i_2,i_3}, \quad [\boldsymbol{T}_2]_{i_3i_2i_1} \leftarrow \lambda_6[\boldsymbol{T}_1]_{i_1,i_2,i_3}$$

5.0.3 Broadcasting

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: $B(k_1)B(k_2) + \sum_{\kappa=1}^{\min(k_1,k_2)} \kappa!\big[\sum_{n_s=0}^{k_1} S(k_1 - n_s)\binom{k_1}{n_s}B(n_s)\big]\big[\sum_{n_b=0}^{k_2} S(k_2 - n_b)\binom{k_2}{n_b}B(n_b)\big]$. The following theorem shows that this 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 efficiently implemented in GPU accelerated numerical libraries such as PyTorch (Paszke et al., 2019) with various primitive tensor operations.

Theorem 2.

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

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 2 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).

5.1 Constructing a Broadcast Tensor for a Specific Partition

Algorithm 1 and 2 detail the steps involved in constructing a permutation equivariant layer between $k_1 \to k_2$'th order permutable tensors. We find it easiest to understand the construction of the broadcast tensors through examples so we provide two more examples before writing the general form of the output rule.

Example 5. Let $\boldsymbol{T}_2 \in \mathbb{R}^{n^2}$ be a 3rd order tensor. Suppose we want to construct a $3 \to 3$ equivariant layer. We consider partitions of $\{1, \dots, 6\}$. 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 1st order permutable tensor: $\sum_{i_3} \boldsymbol{T}_{i_1,i_3,i_3}$ and gets broadcast to $\boldsymbol{T}_2$ as follows:

$$[\boldsymbol{T}_2]_{i_1 i_2 i_2} \leftarrow \lambda \sum_{i_3} [\boldsymbol{T}_1]_{i_1 i_3 i_3}$$

Example 6. Using the same third order tensors in Example (5), 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 $\boldsymbol{T}_1$ to the last index of $\boldsymbol{T}_2$. 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.

$$[\boldsymbol{T}_2]_{i_1 i_1 i_2} \leftarrow \lambda \sum_{i_3} [\boldsymbol{T}_1]_{i_2,i_3,i_3}.$$

In general, if we have $\boldsymbol{T}_1 \in \mathbb{R}^{n^{k_1}}, \boldsymbol{T}_2 \in \mathbb{R}^{n^{k_2}}$, the broadcast tensor associated with partition $\mathcal{P} = \{S_1, \dots, S_{|\mathcal{P}|}\}$, where the S_i's are non-intersecting subsets of $\{1, 2, \dots, k_1 + k_2\}$, can be described elementwise by the following rule:

$$[\boldsymbol{T}_2]_{i_{\psi(1)},\dots,i_{\psi(k_2)}} \leftarrow \sum_{i_z : z \notin \{\psi(j) | j \in [k_2]\}} [\boldsymbol{T}_1]_{i_{\psi(k+1)},\dots,i_{\psi(k_1+k_2)}}$$

for all entries $i_j \in \{1, \dots n\}$, and $i_1 \neq i_2 \neq \dots \neq i_{|\mathcal{P}|}$. The summation is applied over indices that are transfer variables. The function ψ sends each of the numbers $1, 2, \dots, k_1 + k_2$ to the index of the subset containing it. Formally, $\psi : \{1, 2, \dots, k_1 + k_2\} \to \{1, 2, \dots, |\mathcal{P}|\}$, $\psi(a) = b,$ where $a \in S_b$ in the partition $\mathcal{P}$.

5.2 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 were instead $n^{k_1} \times d_1$ we would have an intermediate tensor of shape $n^{k_2} \times d_1 \times B(k_1 + k_2)$ which can subsequently be mixed to $n^{k_2} \times d_2$, where d_2 is the desired output dimension size. Algorithm 2 provides

the pseudocode for a $k_1 \to k_2$ permutation equivariant layer. See the code samples in the Supplement for a concrete instantiation of some of the equivariant layers. 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 $\mathbb{R}^{n^{k_2} \times d_1}$

```
1 lst = [ ]
2 foreach Partition P ∈ Partitions of (k1 + k2) do
3 |   Construct X^P, the appropriate broadcast
  |    tensor of X corresponding to P according to
  |    section 5.1
4 |   lst.append(X^P)
5 end foreach
6 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(k1 + k2) intermediate tensors :
    ops ← construct_ops(k1, k2, X)
2 Stack layers to produce X̂ ∈ R^{n^{k2} × d1 × B(k1+k2)}:
    X̂ ← stack(ops)
3 Linearly mix X̂ with M along X̂'s last two indices:
    Z ← X̂ · M
4 return Z
```

6 EXPERIMENTS

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

6.1 Counting Unique Elements of a Set

We follow the experimental setup proposed by Lee et al. (2019). Using the Omniglot (Lake et al., 2015) dataset which contains 1623 different handwritten characters from 50 different languages, we sample sets of characters and try to predict the number of unique characters in each set.

For each batch in our training data we sample a set size n uniformly from $\{6, 7, \ldots, 10\}$. For each example 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 Deep Sets (Zaheer et al., 2017), and Set Transformer (Lee et al., 2019) based models against our models that used 2nd order equivariant layers. 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. For the second and third order equivariant networks, we construct second and third order permutable tensors by using a second and third order outer product. Following Lee et al. (2019) 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 all experiments, we fixed the architecture of the intermediate convolutional neural net used to map the input image to a fixed d dimensional feature vector. See the supplement for full details on our model architectures and training procedure.

6.2 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 drug combinations of size $1, 2, 3, 4$ and 5 out of eight possible antibiotics. For every set

Table 1: Test accuracy on unique characters task

Model	Accuracy	Parameters	Time per minibatch
Deep Sets	0.62(±0.011)	298, 281	0.121s
Set Transformer	0.74(±0.021)	216, 745	0.128s
2nd Order Network (ours)	0.72(±0.018)	139, 661	0.128s
3rd Order Network (ours)	0.69(±0.014)	96, 425	0.129s

Table 2: Test MAE, RMSE on drug combination efficacy prediction task

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

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 (2) shows the average MAE and RMSE over five random seeds with the standard deviations in parenthesis. The 2nd and third 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. For full details on the experiments, see our supplement.

We find that our permutation equivariant architectures perform comparably to the Set Transformer and Deep Sets baselines on both tasks. While we cannot definitively say that our models work better or worse, our models do require much fewer parameters to achieve similar results.

7 CONCLUSION

We gave a prescription for how to construct permutation equivariant layers between permutable tensors of arbitrary order and showed that the layers constructed in this manner are the full set of equivariant operations possible in each layer. Our experiments demonstrate their effectiveness compared to other permutation equivariant architectures and provide a template of how to use more expressive intermediate permutation equivariant layers.

References

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.

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.

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

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.

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

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.

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.

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.

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.

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.

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

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.

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.

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

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

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

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.

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.

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, 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, 2019.

Yujia Li, Daniel Tarlow, Marc Brockschmidt, and Richard Zemel. Gated graph sequence neural networks. *arXiv preprint arXiv:1511.05493*, 2015.

Haggai Maron, Heli Ben-Hamu, Nadav Shamir, and Yaron Lipman. Invariant and equivariant graph networks. *arXiv preprint arXiv:1812.09902*, 2018.

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.

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.

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.

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`.

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.

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.

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.

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.

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`.

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.

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.

Appendix

- Section A of our appendix gives further details on the permutation equivariant layers and their construction
- Section B discusses the general architectural template for using higher order permutation equivariant layers
- Section C and D provide further details on the experiments

A EQUIVARIANT LAYER DETAILS

For a permutation equivariant layer that maps a 2nd order permutable tensor to a 2nd order permutable tensor, we have $B(2+2) = 15$ operations in total that are listed in Table 3. Notice that the intermediate summation tensors (row sum, colum sum, diagonal) get re-used in multiple operations.

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

A.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.

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

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

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

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

Theorem 3. Linear combinations of kth order permutable tensors of size $\mathbb{R}^{n^k \times 1}$ are still kth 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 kth order permutable tensors of size $\mathbb{R}^{n^k \times 1}$ extends to permutable tensors of arbitrary feature dimension size $\mathbb{R}^{n^k \times d}$.

B ARCHITECTURE

Recall that the layers we propose take in as input a k_1th order permutable tensor and return a k_2th order permutable tensor. In most cases we will want $k_1 = k_2$ and we can view the layer as a linear mapping that can replace row-wise linear layers. If our data comes in the form of kth order permutable tensors then we can simply use them necessary and treat them as a linear layer that is more expressive than a row-wise linear layer. Generally our data *does* come in the fomr of first order permutable tensors. In the set learning case, for each instance of data, we might have a d-dimensional feature vector associated with each entitty. Our data comes in pairs (X_i, y_i), where $X_i = \{x_1^{(i)}, x_2^{(i)}, \dots, x_n^{(i)}\}$ is a set of items. $y_i \in \mathbb{R}$ for a regression task or $y_i \in \{0, 1, \dots, M-1\}$ for a M-way multiclass classfication task. We can trivially construct kth order tensors by taking a k-way outer product of the entities.

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

1. encode each item of the set (ex: a resnet for images, row-wise MLP, row-wise linear layer, etc)
2. construct a 2nd (or 3rd) order tensor through a 2nd(or 3rd) order outer 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
4. use 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 MLP to get the final output

In the models used in our experiments, we used two permutation equivariant layers in step 3 above.

C UNIQUE CHARACTERS EXPERIMENT

In the unique characters experiment, we receive as input a set of images sampled from the Omniglot (Lake et al., 2015) dataset. The goal is to then predict the number of unique characters in this set of images.

Each minibatch of data is generated in the following way:

- Sample a set length n for the batch uniformly at random from $\{6, 7, \ldots, 10\}$ from the Omniglot dataset
- For the ith example of the batch, sample a unique character count $c_i \in \{1, 2, \ldots, n\}$
- Sample exactly n images such that among the n images, there are exactly c_i unique characters for the ith example of the batch.

C.1 Training Details

We follow the same experimental procedure as Lee et al. (2019); Shi et al. (2020) and perform Poisson regression by having the network return a 1-dimensional output for the λ parameter of the predicted Poisson distribution. We use the log likelihood as the loss function to optimize over during training.

We trained the models with a fixed batch size of 32, and a constant learning rate of 10^{-3} using ADAM as the optimizer (Kingma and Ba, 2014). We had experimented with learning rates between 10^{-4} and 10^{-3} and found that the learning rate did not meaningfully affect the results. For all models, we trained for at most $200,000$ minibatches. We used the same basic CNN encoder for all the models we experimented with. The architecture used for the CNN is the same one used by Shi et al. (2020) except with the number of channels set to 12 in each of the convolutional layers. Using additional channels, and additional layers in the CNN embedding module did not affect the prediction accuracy much so we kept the CNN embedding module's architecture the same for all experiments.

The main hyperparameters to tune were the embedding dimension of the CNN (also the input dimension of the permutation equivariant layers) and the output dimension of the permutation equivariant layer. We tested embedding and output dimension sizes from $32, 64, 96, 128$, and 256 by training for 1000 minibatches and using the embedding/output size settings that had the best training accuracy up to that point.

Table 2 details the misceallaenous hyperparameter settings. All experiments were run on an Nvidia GeForce GTX 1080 Ti GPU.

Table 4: Misc. hyperparameters for unique characters experiments

Model	Batch Size	Dropout	Embedding dim	Output 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

D DRUG COMBINATION EXPERIMENT

In the drug combinations experiment, we receive as input a set of drugs that was applied to inhibit the growth of pathogenic ecoli. The goal is to then predict the efficacy of this drug combination in terms of the proportion of growth exhibited by the specimen compared to a control experiment (where no drugs were applied). We used the data of drug combination experiments on E.coli released by Tekin et al. (2018). The data is originally released in pdf format so we parsed the pdf to csv format to extract the data. As mentioned in the main body of the paper, the training data consisted of: all drug experiments with up to 4 drugs and 80% of the experiments on 5 drugs. The remaining 20% of five drug experiment data was used as the test set.

We zero pad the data so that each batch of data comes in as a tuple of int tensor and float tensor of uniform shape: $(b \times 5, b \times 5)$, where b is the batch size. The int tensor contains the identities of each drug used in the experiment (expressed as an integer between 0 and 8). The float tensor contains the dosage levels of each drug used.

D.1 Training Details

We trained all models for at most $12,000$ epochs, with a batch size of 512, and a learning rate 10^{-3} using ADAM (Kingma and Ba, 2014). Our models minimized mean squared error loss.

The main parameter to tune here, as in the unique characters experiment, were the input and output dimensions (denoted hidden size and output size in Table 3) of the permutation equivariant layers for our models.

Before honing in on a specific architecture that used our 2nd and 3rd order equivariant layers, we tested other architectures that used more equivariant layers, or a multilayer perceptron (instead of a linear layer) output on shorter training schedules (around 1000 epochs). We found that using dropout (with $p = 0.50$) was crucial to avoid overfitting to the training set in our 2nd and 3rd order networks. We also tried using batch normalization but found that it only slowed training without offering much if any improvements in training/test error.

Table 5: Misc. hyperparameters for drug combination efficacy experiments

Model	Batch Size	Dropout	Embed dim	Hidden Dimension	Output dim
Deep Sets	512	0	32	512	512
Set Transformer	512	0	32	256	512
2nd Order Network (ours)	512	0.50	32	256	512
3nd Order Network (ours)	512	0.50	32	256	256

Third order networks are still a bit hard to train and take a bit more time than the 2nd order network. This is not surprising since the forward pass of the third order network requires an expensive 3rd order outer product.

E CODE

For the full implementation of our model and experiments, please see our repository: `https://github.com/horacepan/permeqlayers/`.