JANOSSY POOLING: LEARNING DEEP PERMUTATION-INVARIANT FUNCTIONS FOR VARIABLE-SIZE INPUTS

Anonymous authors
Paper under double-blind review

ABSTRACT

We consider a simple and overarching representation for permutation-invariant functions of sequences (or set functions). Our approach, which we call Janossy pooling, expresses a permutation-invariant function as the average of a permutation-sensitive function applied to all reorderings of the input sequence. This allows us to leverage the rich and mature literature on permutation-sensitive functions to construct novel and flexible permutation-invariant functions. If carried out naively, Janossy pooling can be computationally prohibitive. To allow computational tractability, we consider three kinds of approximations: canonical orderings of sequences, functions with k-order interactions, and stochastic optimization algorithms with random permutations. Our framework unifies a variety of existing work in the literature, and suggests possible modeling and algorithmic extensions. We explore a few in our experiments, which demonstrate improved performance over current state-of-the-art methods.

1 INTRODUCTION

Pooling is a fundamental operation in deep learning architectures (LeCun et al., 2015). The role of pooling is to merge a collection of related features into a single, possibly vector-valued, summary feature. A prototypical example is in convolutional neural networks (CNNs)(LeCun et al., 1995), where linear activations of features in neighborhoods of image locations are pooled together to construct more abstract features. A more modern example is in neural networks for graphs, where each layer pools together embeddings of neighbors of a vertex to form a new embedding for that vertex, see for instance, (Kipf & Welling, 2016; Atwood & Towsley, 2016; Hamilton et al., 2017; Velickovic et al., 2017; Monti et al., 2017; Xu et al., 2018; Liu et al., 2018; Liben-Nowell & Kleinberg, 2007; van den Berg et al., 2017; Duvenaud et al., 2015; Gilmer et al., 2017; Ying et al., 2018).

A common requirement of a pooling operator is invariance to the ordering of the input features. In CNNs for images, this allows invariance to translations of objects, while for graphs, this allows invariance to relabeling of graph vertices. Existing pooling operators are mostly limited to predefined heuristics such as max-pool, min-pool, sum, or average. Another desirable characteristic of pooling layers is the ability to take variable-size inputs. This is less important in images, where neighborhoods are usually fixed *a priori*. However in applications involving graphs, the number of neighbors of different vertices can vary widely. Our goal is to design flexible and learnable pooling operators satisfying these two desiderata.

Abstractly, we will view pooling as a permutation-invariant (or symmetric) function acting on finite but arbitrary length sequences $\boldsymbol{h}$. All elements h_i of the sequences are features belonging to some space $\mathbb{H}$ (which itself could be a high-dimensional Euclidean space $\mathbb{R}^n$ or some subset thereof, $n > 0$). The sequences $\boldsymbol{h}$ are elements of the union of products of the $\mathbb{H}$-space: $\boldsymbol{h} \in \bigcup_{j=0}^{\infty} \mathbb{H}^j$. Throughout the paper, we will use Π_n to represent the set of all permutations of the integers 1 to n, where n will often be clear from the context. In particular, $\boldsymbol{h}_\pi$, $\pi \in \Pi_{|\boldsymbol{h}|}$, will represent a reordering of the elements of a sequence $\boldsymbol{h}$ according to π, where $|\boldsymbol{h}|$ is the length of the sequence $\boldsymbol{h}$. We will use the double bar superscript $\overline{\overline{f}}$ to indicate that a function is permutation invariant, returning the same value no matter the order of its arguments: $\overline{\overline{f}}(\boldsymbol{h}) = \overline{\overline{f}}(\boldsymbol{h}_\pi), \forall \pi \in \Pi_{|\boldsymbol{h}|}$. We will use the arrow superscript $\vec{f}$ to indicate general functions on sequences $\boldsymbol{h}$ which may or may not be permutation-invariant. Functions f without any markers are 'simple' functions, acting on elements in $\mathbb{H}$, scalars or any other argument that is not a sequence of elements in $\mathbb{H}$.

Our goal in this paper is to model and learn permutation-sensitive functions $\vec{f}$ that can be used to construct flexible and learnable permutation-invariant neural networks. A recent step in this direction is work on *deep sets* by Zaheer et al. (2017), who argued for learning permutation-invariant functions through the following composition:

$$\overline{\overline{y}}(\boldsymbol{x};\boldsymbol{\theta}^{(\rho)},\boldsymbol{\theta}^{(f)},\boldsymbol{\theta}^{(h)}) = \rho\left(\overline{\overline{f}}(|\boldsymbol{h}|,\boldsymbol{h};\boldsymbol{\theta}^{(f)});\boldsymbol{\theta}^{(\rho)}\right), \text{where} \tag{1}$$

$$\overline{\overline{f}}(|\boldsymbol{h}|,\boldsymbol{h};\boldsymbol{\theta}^{(f)}) = \frac{1}{|\boldsymbol{h}|}\sum_{j=1}^{|\boldsymbol{h}|} f(h_j;\boldsymbol{\theta}^{(f)}) \quad \text{and} \quad \boldsymbol{h} \equiv h(\boldsymbol{x};\boldsymbol{\theta}^{(h)}). \tag{2}$$

Here (a) $\boldsymbol{x} \in \mathbb{X}$ is the training data, $\boldsymbol{h} \in \mathbb{H}$ is the embedding space of h, and $h : \mathbb{X}\times\mathbb{R}^a \to \bigcup_{j=0}^{\infty}\mathbb{H}^j$, $a > 0$ is the embedding given by lower layers of the neural network with parameters $\boldsymbol{\theta}^{(h)} \in \mathbb{R}^a$, and (b) $f : \mathbb{H}\times\mathbb{R}^b \to \mathbb{F}$ is a middle-layer embedding function with parameters $\boldsymbol{\theta}^{(f)} \in \mathbb{R}^b$, $b > 0$, and $\mathbb{F}$ is the embedding (output) space of f; and (c) $\rho : \mathbb{F}\times\mathbb{R}^c \to \mathbb{Y}$ is a neural network with parameters $\boldsymbol{\theta}^{(\rho)} \in \mathbb{R}^c$, $c > 0$, that maps to the final output space $\mathbb{Y}$. Typically $\mathbb{H}$ and $\mathbb{F}$ are high-dimensional real-valued spaces; $\mathbb{Y}$ is often $\mathbb{R}^d$ in d-dimensional regression problems or the simplex in classification problems. Effectively, the neural network f learns an embedding for each element in $\mathbb{H}$, and given a sequence $\boldsymbol{h}$, its component embeddings are added together before a second neural network transformation ρ is applied. Note that the function h may be the identity mapping $h(\boldsymbol{x};\cdot) = \boldsymbol{x}$ that makes $\overline{\overline{f}}$ act directly on the input data. Zaheer et al. (2017) argues that, under reasonable assumptions, the above architecture with suitably flexible functions f and ρ is capable of approximating any symmetric function on $\boldsymbol{h}$-sequences, which justifies the widespread use of average (sum) pooling to make neural networks permutation invariant in Duvenaud et al. (2015), Hamilton et al. (2017), Kipf & Welling (2016), Atwood & Towsley (2016), Niepert et al. (2016), among other works.

In practice, however, there is a gap between flexibility and learnability. While the architecture of equations 1 and 2 is a universal approximator to all permutation-invariant functions, it does not allow structural knowledge about the function $\overline{\overline{y}}$ to be encoded easily. Consider for instance trying to learn the permutation-invariant function $\overline{\overline{y'}}(\boldsymbol{x}) = \max_{i,j\leq|\boldsymbol{x}|}|x_i - x_j|$. With such higher-order interactions between the elements of $\boldsymbol{h}$, the functions $\vec{f}$ of equation 2 cannot capture any useful intermediate representations towards the final output, with the burden shifted entirely to the function ρ. Learning ρ means learning to undo mixing performed by the summation layer $\overline{\overline{f}}(|\boldsymbol{h}|,\boldsymbol{h};\boldsymbol{\theta}^{(f)}) = \frac{1}{|\boldsymbol{h}|}\sum_{j=1}^{|\boldsymbol{h}|} f(h_j;\boldsymbol{\theta}^{(f)})$. Unfortunately, as we show in our experiments, in many applications this is too much to ask of ρ.

Contributions. In this work we investigate a learnable permutation-invariant pooling layer for variable-size inputs inspired by the Janossy density framework (widely used in the theory of point processes (Daley & Vere-Jones, 2003, Chapter 7)). This approach, which we call Janossy pooling, directly allows the user to model what higher-order dependencies in $\boldsymbol{h}$ are relevant in the pooling.

Figure 1 summarizes a neural network with a single pooling layer $\overline{\overline{f}}$: given an input embedding $\boldsymbol{h}$, we apply a learnable (permutation-sensitive) function $\vec{f}$ to every permutation $\boldsymbol{h}_\pi$ of the input sequence $\boldsymbol{h}$. These outputs are added together, and fed to the second function ρ. Examples of function $\vec{f}$ include feedfoward and recurrent neural networks. We call the operation used to construct $\overline{\overline{f}}$ from $\vec{f}$ the *Janossy pooling*. A more detail description of Janossy

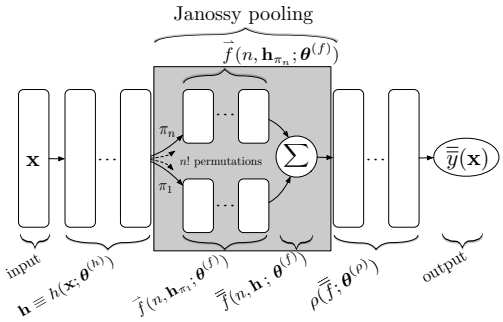

Figure 1: A neural network with a single Janossy pooling layer. The embedding $\boldsymbol{h}$ is permuted in all $|\boldsymbol{h}|!$ possible ways, and for each permutation $\boldsymbol{h}_\pi$, $\vec{f}(|\boldsymbol{h}|,\boldsymbol{h}_\pi;\boldsymbol{\theta}^{(f)})$ is computed. These results are added together, and the result is passed to a second function ρ parameterized by $\boldsymbol{\theta}^{(\rho)}$, which gives the final permutation-invariant output $\overline{\overline{y}}(\boldsymbol{x};\boldsymbol{\theta}^{(h)},\boldsymbol{\theta}^{(f)},\boldsymbol{\theta}^{(\rho)})$. We discuss how this scheme can be made computationally tractable.

pooling is given in Definition 2.1. Further, we contribute the following analysis:

(a) We show *deep sets* (Zaheer et al., 2017) is a special case of Janossy pooling if the function $\vec{f}$ depends only on the first element of the vector $\boldsymbol{h}_\pi$. In the most general form of Janossy pooling (as described above), $\vec{f}$ depends on its entire input sequence $\boldsymbol{h}_\pi$. This naturally raises the possibility of intermediate choices of $\vec{f}$ that allow practitioners to trade between flexibility and tractability. We will show that functions $\vec{f}$ that depend on their first k arguments of $\boldsymbol{h}_\pi$ allow the Janossy pooling layer to capture up to k-ary dependencies in $\boldsymbol{h}$.

(b) We show Janossy pooling can be used to learn permutation-invariant neural networks $\overline{\overline{y}}(\boldsymbol{x})$ by sampling a random permutation during training, and then modeling this permutation using a sequence model such as a recurrent neural network (LSTMs (Hochreiter & Schmidhuber, 1997), GRUs (Cho et al., 2014)) or a vector model such as a feedforward network. We call this permutation-sampling learning algorithm π-SGD (π-Stochastic Gradient Descent). Our analysis explains why this seemly unsound procedure is theoretically justified, which sheds light on the recent puzzling success of permutation sampling and LSTMs in relational models Moore & Neville (2017); Hamilton et al. (2017). We show that this property relates to what we randomized model ensemble techniques.

(c) In Zaheer et al. (2017), the authors describe a connection between *deep sets* and infinite de Finetti exchangeabilty. We provide a probabilistic connection between Janossy pooling and *finite* de Finetti exchangeabilty (Diaconis, 1977).

2 JANOSSY POOLING

In this section we formalize the Janossy pooling function $\overline{\overline{f}}$. We start with a function $\vec{f}$, parameterized by $\boldsymbol{\theta}^{(f)}$, which can take any variable-size sequence as input: a sequence of matrices (such as a sequence of images), a sequence of vectors (such as a sequence of vector embeddings), or a variable-size sequence of features or embeddings representing the neighbors of a node in an attributed graph. In practice, we implement $\vec{f}$ with a neural network. Formalizing Figure 1 from Section 1, we use $\vec{f}$ to define $\overline{\overline{f}}$:

Definition 2.1: [Janossy pooling] Consider a function $\vec{f} : \mathbb{N} \times \bigcup_{j=0}^{\infty} \mathbb{H}^j \times \mathbb{R}^b \to \mathbb{Y}$ on variable-length but finite sequences $\boldsymbol{h}$, parameterized by $\boldsymbol{\theta}^{(f)} \in \mathbb{R}^b$, $b > 0$. A permutation-invariant function $\overline{\overline{f}} : \bigcup_{j=0}^{\infty} \mathbb{H}^j \to \mathbb{Y}$ is the Janossy function associated with $\vec{f}$ if

$$\overline{\overline{f}}(\boldsymbol{h}; \boldsymbol{\theta}^{(f)}) = \frac{1}{|\boldsymbol{h}|!} \sum_{\pi \in \Pi_{|\boldsymbol{h}|}} \vec{f}(|\boldsymbol{h}|, \boldsymbol{h}_\pi; \boldsymbol{\theta}^{(f)}), \tag{3}$$

where $\Pi_{|\boldsymbol{h}|}$ is the set of all permutations of the integers 1 to $|\boldsymbol{h}|$, and $\boldsymbol{h}_\pi$ represents a particular reordering of the elements of sequence $\boldsymbol{h}$ according to $\pi \in \Pi_{|\boldsymbol{h}|}$. We denote Janossy pooling the operation used to construct $\overline{\overline{f}}$ from $\vec{f}$. $\Diamond$

Definition 2.1 provides a conceptually simple approach to constructing permutation-invariant functions from arbitrary sequence-functions. If $\vec{f}$ is a vector-valued function, then so is $\overline{\overline{f}}$, and in practice, one might pass this vector output of $\overline{\overline{f}}$ through a second function ρ (e.g. a neural network parameterized by $\theta^{(\rho)}$):

$$\overline{\overline{y}}(\boldsymbol{h}; \boldsymbol{\theta}^{(f)}, \boldsymbol{\theta}^{(p)}) = \rho\left(\frac{1}{|\boldsymbol{h}|!} \sum_{\pi \in \Pi_{|\boldsymbol{h}|}} \vec{f}(|\boldsymbol{h}|, \boldsymbol{h}_\pi; \boldsymbol{\theta}^{(f)}); \boldsymbol{\theta}^{(p)}\right). \tag{4}$$

Theoretically, equation 3 can capture any permutation-invariant function for a flexible-enough family of functions for $\vec{f}$, thus, at least theoretically, ρ in equation 4 provides no additional representational power. In practice, ρ can improve learnability by capturing common aspects across all terms in the summation. Furthermore, when we look at approximations to equation 3, adding ρ can help recover some of the lost model capacity. Overall then, equation 4 represents one layer of Janossy pooling, forming a constituent part of a bigger neural network. Figure 1 summarizes this.

Janossy pooling, as defined in equation 3 and 4 is intractable; the computational cost of summing over all permutations (for prediction), and backpropagating gradients (for learning) is likely prohibitive for most problems of interest. Nevertheless, it provides an overarching framework to unify existing methods, and to extend them. In what follows we present strategies for mitigating this computational cost.

2.1 Tractability through Canonical Input Orderings

A simple way to achieve permutation-invariance without the summation in equation 3 is to order the elements of $\boldsymbol{h}$ according to some canonical ordering based on their values, and then feed the reordered sequence to $\vec{f}$. This way, all the permutations of $\boldsymbol{h}$ are mapped to the same canonical sequence. Effectively, this imposes the following structural constraint on $\vec{f}$: $\vec{f}(\boldsymbol{h})$ equals 0 for all sequences that do not follow the canonical permutation. This allows us to use more complex $\vec{f}$ models, such as LSTMs and GRUs, that can capture arbitrary relationships in the canonical ordering of $\boldsymbol{h}$.

Examples of the canonical ordering approach already exist in the literature. We first describe Niepert et al. (2016), which orders nodes in a graph according to a user-specified ranking such as betweenness centrality (say from high to low). This approach is useful only if the canonical ordering is relevant to the task at hand. Niepert et al. acknowledges this shortcoming by showing an example where a poor vertex traversal order significantly degrades classification accuracy. As an idealized example, consider input sequences $\boldsymbol{h} = \big((h_{i,1}, h_{i,2})\big)_{i=1}^{n}$, with $(h_{i,1}, h_{i,2}) \in \mathbb{H} = \mathbb{R}^2$, and components $x_{i,1}$ and $x_{i,2}$ sampled independently of each other. Choosing a canonical ordering to sort $\boldsymbol{h}$ according to $h_{\cdot,1}$ when the task at hand depends on a canonical ordering based on $h_{\cdot,2}$ can lead to poor prediction accuracy.

Rather than pre-defining a good canonical order, one can try to learn it from the data. This requires searching over the discrete space of all $|\boldsymbol{h}|!$ permutations of the input vector $\boldsymbol{h}$. In practice, this discrete optimization relies on heuristics (Vinyals et al., 2016; Rezatofighi et al., 2018). Alternatively, instead of choosing a single canonical ordering, one can choose multiple orderings, resulting in ensemble methods that average across multiple permutations. These can be viewed as more refined (possibly data-driven) approximations to equation 3.

2.2 Tractability through k-ary dependencies

Here, we provide a different spectrum of options to trade-off flexibility, complexity, and generalizability in Janossy pooling. Now, to simplify the sum over permutations in equation 3, we impose structural constraints where $\vec{f}(\boldsymbol{h})$ depends only on the first k elements of its input sequence. This amounts to the assumption that only k-ary dependencies in $\boldsymbol{h}$ are relevant to the task at hand.

Definition 2.2: [k-ary Janossy pooling] Fix $k \in \mathbb{N}$. For any sequence $\boldsymbol{h}$, define $\downarrow_k(\boldsymbol{h})$ as its projection to a length k sequence; in particular, if $|\boldsymbol{h}| \geq k$, we keep the first k elements, while if $|\boldsymbol{h}| < k$, we append $(k - |\boldsymbol{h}|)$ zeros at the end of $\boldsymbol{h}$ to make it at least of size k. Then, a k-ary permutation-invariant Janossy function $\overline{\vec{f}}$ is given by

$$\overline{\vec{f}}(|\boldsymbol{h}|, \boldsymbol{h}; \boldsymbol{\theta}^{(f)}) = \frac{1}{|\boldsymbol{h}|!} \sum_{\pi \in \Pi_{|\boldsymbol{h}|}} \vec{f}(k, \downarrow_k(\boldsymbol{h}_\pi); \boldsymbol{\theta}^{(f)}). \tag{5}$$

$\Diamond$

Proposition 2.1 shows that if $|\boldsymbol{h}| > k$, equation 5 only needs to sum over $|\boldsymbol{h}|!/(|\boldsymbol{h}| - k)!$ terms.

Proposition 2.1. *The Janossy pooling in equation 5 requires summing over only $\frac{n!}{(n-k)!}$ terms, thus saving computation when $k < |\boldsymbol{h}|$.*

Proof. Without loss of generality assume $|\boldsymbol{h}| \geq k$, we can pad $\boldsymbol{h}$ with zeros if needed. Define two permutations $\pi, \pi' \in \Pi_{|\boldsymbol{h}|}$ that agree on the first k elements as *k-equivalent*. Such permutations satisfy $\vec{f}(k, \downarrow_k(\boldsymbol{h}_\pi); \boldsymbol{\theta}^{(f)}) = \vec{f}(k, \downarrow_k(\boldsymbol{h}_{\pi'}); \boldsymbol{\theta}^{(f)})$. These two permutations belong to the same equivalence class, containing a total of $(|\boldsymbol{h}| - k)!$ permutations (obtained by permuting the last

$(|\boldsymbol{h}| - k)$ elements). Overall, we then have a total of $|\boldsymbol{h}|!/(|\boldsymbol{h}| - k)!$ equivalence classes. Write the set of equivalence classes as $\Pi^k_{|\boldsymbol{h}|}$, and represent each by one of its elements. Then,

$$\begin{aligned}\overline{\overline{f}}_k(\boldsymbol{h}; \boldsymbol{\theta}^{(f)}) &= \frac{1}{|\boldsymbol{h}|!} \sum_{\pi \in \Pi_{|\boldsymbol{h}|}} \vec{f}(k, \downarrow_k(\boldsymbol{h}_\pi); \boldsymbol{\theta}^{(f)}) \\ &= \frac{(|\boldsymbol{h}| - k)!}{|\boldsymbol{h}|!} \sum_{\pi \in \Pi^k_{|\boldsymbol{h}|}} \vec{f}\big(k, \downarrow_k(\boldsymbol{h}_\pi)); \boldsymbol{\theta}^{(f)}\big) \qquad (6)\end{aligned}$$

is now a summation over only $|\boldsymbol{h}|!/(|\boldsymbol{h}| - k)!$ terms. □

The sum in equation 6 can be written equivalently as $\sum_{i_1, i_2, \ldots, i_k \in \mathbb{I}_{|\boldsymbol{h}|}} \vec{f}\big(k, \boldsymbol{h}_{i_1, i_2 \ldots, i_k}; \boldsymbol{\theta}^{(f)}\big)$, where $\mathbb{I}_n$ is the set of all permutations of $\{1, 2, \ldots, n\}$ taken k at a time. Note that the value of k balances computational savings and model flexibility; it can be selected as a hyperparameter based on a-priori beliefs or through typical hyperparameter tuning strategies.

Remark 2.1 (Deep sets (Zaheer et al., 2017) is a 1-ary (unary) Janossy pooling)**.** *Equation 6 represented with $k = 1$ yields the model $\rho\big(\sum_{i=1}^{|\boldsymbol{h}|} \vec{f}(h_i; \boldsymbol{\theta}^{(f)}); \boldsymbol{\theta}^{(\rho)}\big)$. Noting that $\vec{f}$ only takes a single input, we can redefine it and recover equation 2.*

Not surprisingly, the computational savings obtained from k-ary Janossy pooling come at the cost of reduced model flexibility. The next result formalizes this.

Theorem 2.1. *For any $k \in \mathbb{N}$, define $\mathcal{F}_k$ as the set of all permutation invariant functions that can be represented by Janossy pooling with k-ary dependencies. Then, $\mathcal{F}_{k-1}$ is a* proper *subset of $\mathcal{F}_k$ if the space $\mathbb{H}$ is not trivial (i.e. if the cardinality of $\mathbb{H}$ is greater than 1). Thus, Janossy pooling with k-ary dependencies can express any Janossy pooling function with $(k-1)$-ary dependencies, but the converse does not hold.*

Proof.
($\mathcal{F}_{k-1} \subset \mathcal{F}_k$): Consider any element $\overline{\overline{f}}_{k-1} \in \mathcal{F}_{k-1}$. For any order-dependent function $\vec{f}$, define $\vec{f}_{\downarrow k}(\boldsymbol{h}) = \vec{f}(\downarrow_k(\boldsymbol{h}))$. Then, by definition, there exists a function $\vec{f}$ such that

$$\overline{\overline{f}}_{k-1}(\boldsymbol{h}; \boldsymbol{\theta}^{(f)}) = \frac{1}{|\boldsymbol{h}|!} \sum_{\pi \in \Pi_{|\boldsymbol{h}|}} \vec{f}(\downarrow_{k-1}(\boldsymbol{h}_\pi); \boldsymbol{\theta}^{(f)}) = \frac{1}{|\boldsymbol{h}|!} \sum_{\pi \in \Pi_{|\boldsymbol{h}|}} \vec{f}_{\downarrow_{k-1}}(\boldsymbol{h}_\pi; \boldsymbol{\theta}^{(f)}) \qquad (7)$$

$$= \frac{1}{|\boldsymbol{h}|!} \sum_{\pi \in \Pi_{|\boldsymbol{h}|}} \vec{f}_{\downarrow_{k-1}}(\downarrow_k(\boldsymbol{h}_\pi); \boldsymbol{\theta}^{(f)}) \qquad (8)$$

($\mathcal{F}_k \not\subset \mathcal{F}_{k-1}$): Here, we need to construct a function $\overline{\overline{f}} \in \mathcal{F}_k$ that is not present in $\mathcal{F}_{k-1}$. The construction is given in the Supplementary Material. □

Corollary 2.1. *For $k > 1$, the* deep sets *function in equation 1 (Zaheer et al., 2017) pushes the modeling of k-ary relationships to ρ.*

Proof. Deep sets functions can be expressed via Janossy pooling with $k = 1$. Thus, by Theorem 2.1, $\overline{\overline{f}}$ in equation 2 cannot express all functions that can be expressed by higher-order (i.e. $k > 1$) Janossy pooling operations. Consequently, if the *deep sets* function can express a premutation-invariant function, the expressive power must have been pushed to ρ. □

2.3 Tractability through Permutation Sampling

Another approach to tractable Janossy pooling uses *random* permutations of the input $\boldsymbol{h}$ during training. Like the approach in Section 2.1, this offers significant computational savings, allowing more complex models for $\vec{f}$ such as LSTMs and GRUs. Unlike that earlier approach, this is considerable more flexible, avoiding the need to learn a canonical-ordering or making assumptions about the dependencies between the elements of $\boldsymbol{h}$ and the objective function. The approach of sampling random permutations has been previously used in relational learning tasks (Moore & Neville, 2017; Hamilton et al., 2017) as a heuristic with an LSTM as $\vec{f}$. Both these papers report that permutation sampling outperforms or closely marches all other tested neural network models they tried. We provide a theoretical framework to understand and extend such approaches.

For simplicity, we will focus on a single Janossy pooling layer and a single sampled permutation, although generalizing these conditions is easy. We assume a supervised learning setting, though our analysis easily extends to unsupervised learning. We are given training data $\mathcal{D} \equiv \{(\boldsymbol{x}_1, \boldsymbol{y}_1), \ldots, (\boldsymbol{x}_N, \boldsymbol{y}_N)\}$, where $\boldsymbol{y}_i \in \mathbb{Y}$ is the target output and $\boldsymbol{x}_i$ its corresponding input. Our original goal was to minimize the empirical loss

$$\overline{\overline{L}}(\mathcal{D}; \boldsymbol{\theta}^{(\rho)}, \boldsymbol{\theta}^{(f)}, \boldsymbol{\theta}^{(h)}) = \frac{1}{N} \sum_{i=1}^{N} L\left(\boldsymbol{y}_i, \rho\Big(\overline{\overline{f}}(\boldsymbol{h}^{(i)}; \boldsymbol{\theta}^{(f)}); \boldsymbol{\theta}^{(\rho)}\Big)\right), \tag{9}$$

where

$$\overline{\overline{f}}(\boldsymbol{h}^{(i)}; \boldsymbol{\theta}^{(f)}) = \frac{1}{|\boldsymbol{h}^{(i)}|!} \sum_{\pi \in \Pi_{|\boldsymbol{h}^{(i)}|}} \vec{f}(|\boldsymbol{h}^{(i)}|, \boldsymbol{h}^{(i)}_{\pi}; \boldsymbol{\theta}^{(f)}) \tag{10}$$

and $\boldsymbol{h}^{(i)} = h(\boldsymbol{x}_i; \boldsymbol{\theta}^{(h)}) \in \cup_{j=0}^{\infty} \mathbb{H}^j$. Computing the gradient of equation 9 is intractable for large inputs $\boldsymbol{h}^{(i)}$, as the backpropagation computation graph branches out for every permutation in the sum. To address this computational challenge, we will turn out attention to stochastic optimization.

Permutation sampling. Consider replacing the Janossy sum in equation 10 with the estimate

$$\hat{\overline{\overline{f}}}(\boldsymbol{h}; \boldsymbol{\theta}^{(f)}) = \vec{f}(|\boldsymbol{h}|, \boldsymbol{h}_{\mathbf{s}}; \boldsymbol{\theta}^{(f)}),$$

where $\mathbf{s}$ is a random permutation sampled uniformly, $\mathbf{s} \sim \text{Unif}(\Pi_{|\boldsymbol{h}|})$. The estimator in equation 2.3 is unbiased: $E_{\mathbf{s}}[\hat{\overline{\overline{f}}}(\boldsymbol{h}; \boldsymbol{\theta}^{(f)})] = \overline{\overline{f}}(\boldsymbol{h}; \boldsymbol{\theta}^{(f)})$. Note however that when $\overline{\overline{f}}$ is chained with another nonlinear function ρ and/or nonlinear loss L, the composition is no longer unbiased: $E_{\mathbf{s}}[L(\boldsymbol{y}, \rho(\vec{f}(|\boldsymbol{h}_{\mathbf{s}}|, \boldsymbol{h}_{\mathbf{s}}; \boldsymbol{\theta}^{(f)}); \boldsymbol{\theta}^{(\rho)}))] \neq L(\boldsymbol{y}, \rho(E_{\mathbf{s}}[\vec{f}(|\boldsymbol{h}_{\mathbf{s}}|, \boldsymbol{h}_{\mathbf{s}}; \boldsymbol{\theta}^{(f)})]; \boldsymbol{\theta}^{(\rho)}))$. Using this estimate we propose the following stochastic approximation algorithm for gradient descent:

Definition 2.3: [π-SGD] Let $\mathcal{B} = \{(\boldsymbol{x}_1, \boldsymbol{y}_1), \ldots, (\boldsymbol{x}_B, \boldsymbol{y}_B)\}$ be a mini-batch i.i.d. sampled uniformly from the training data $\mathcal{D}$. At step t, consider the stochastic gradient descent update

$$\boldsymbol{\theta}_t = \boldsymbol{\theta}_{t-1} - \eta \mathbf{Z}_t, \tag{11}$$

where $\mathbf{Z}_t$ is the random gradient

$$\mathbf{Z}_t = \frac{1}{B} \sum_{i=1}^{B} \nabla_{\boldsymbol{\theta}} L\left(\boldsymbol{y}_i, \rho\Big(\vec{f}(|\boldsymbol{h}^{(i)}|, \boldsymbol{h}^{(i)}_{\mathbf{s}_i}; \boldsymbol{\theta}^{(f)}_t); \boldsymbol{\theta}^{(\rho)}_t\Big)\right)$$

where $\boldsymbol{h}^{(i)} = h(\boldsymbol{x}_i; \boldsymbol{\theta}^{(h)}_t)$, $\boldsymbol{\theta} = (\boldsymbol{\theta}^{(\rho)}, \boldsymbol{\theta}^{(f)}, \boldsymbol{\theta}^{(h)})$, with the random permutations $\{\mathbf{s}_i\}_{i=1}^{B}$, $\mathbf{s}_i \sim \text{Uniform}(\Pi_{|\boldsymbol{h}^{(i)}|})$; the learning rate is $\eta_t \in (0, 1)$ s.t. $\sum_{t=1}^{\infty} \eta_t = \infty$ and $\sum_{t=1}^{\infty} \eta_t^2 < \infty$. $\Diamond$

Effectively, this is a Robbins-Monro stochastic approximation algorithm of gradient descent (Robbins & Monro, 1951; Bottou & Cun, 2004) and optimizes the following modified objective function:

$$\begin{aligned}
\overline{\overline{J}}(\mathcal{D}; \boldsymbol{\theta}^{(\rho)}, \boldsymbol{\theta}^{(f)}, \boldsymbol{\theta}^{(h)}) &= \frac{1}{N} \sum_{i=1}^{N} E_{\mathbf{s}_i}\left[L\left(\boldsymbol{y}_i, \rho\Big(\vec{f}(|\boldsymbol{h}^{(i)}|, \boldsymbol{h}^{(i)}_{\mathbf{s}_i}; \boldsymbol{\theta}^{(f)}); \boldsymbol{\theta}^{(\rho)}\Big)\right)\right] \\
&= \frac{1}{N} \sum_{i=1}^{N} \frac{1}{|\boldsymbol{h}^{(i)}|!} \sum_{\pi \in \Pi_{|\boldsymbol{h}^{(i)}|}} L\left(\boldsymbol{y}_i, \rho\Big(\vec{f}(|\boldsymbol{h}^{(i)}|, \boldsymbol{h}^{(i)}_{\pi}; \boldsymbol{\theta}^{(f)}); \boldsymbol{\theta}^{(\rho)}\Big)\right),
\end{aligned} \tag{12}$$

where $\boldsymbol{h}^{(i)}_{\pi} \equiv (h(\boldsymbol{x}_i; \boldsymbol{\theta}^{(h)}))_{\pi}$ for a permutation π. Observe that the expectation over permutations is now outside the L and ρ functions. Like equation 9, the loss in equation 12 is also permutation invariant. In general though, the optima of $\overline{\overline{J}}$ are different from those of the original objective function $\overline{\overline{L}}$. In the setting where ρ is the identity function (equation 3), and L is convex, the function $\overline{\overline{J}}$ is an upper bound to $\overline{\overline{L}}$ (via Jensen's inequality), and can be viewed as a tractable surrogate to the original Janossy objective. While this property no longer holds for general ρ, our results show that this holds for a composition $\vec{f'} \equiv \rho \circ \vec{f}$ and still exhibits competitive results. In general, increasing the number of sampled permutations decreases bias and variance, and we recover the exact algorithm when all $|\boldsymbol{h}|!$ permutations are sampled. The next result provides some insight into the covergence properties of our algorithm.

Proposition 2.2 (π-SGD Convergence)**.** *Consider the π-SGD algorithm in Definition 2.3. If*

(a) there exists a constant G such that for all $\boldsymbol{\theta}$, $\boldsymbol{G}_t^T \boldsymbol{\theta} \leq G\|\theta - \theta^\star\|_2^2$, where $\boldsymbol{G}_t$ is the true gradient for the full batch over all permutations, $\boldsymbol{G}_t = \nabla_{\boldsymbol{\theta}} \overline{\overline{J}}(\mathcal{D}; \boldsymbol{\theta}_t^{(\rho)}, \boldsymbol{\theta}_t^{(f)}, \boldsymbol{\theta}_t^{(h)})$, where $\boldsymbol{\theta} \equiv (\boldsymbol{\theta}^{(\rho)}, \boldsymbol{\theta}^{(f)}, \boldsymbol{\theta}^{(h)})$.

(b) there exists a constant $\delta > 0$ such that for all $\boldsymbol{\theta}$, $E_t[\|\mathbf{Z}_t\|_2^2] \leq \delta^2(1 + \|\theta - \theta^\star\|_2^2)$, where the expectation is taken with respect to all the data prior to step t.

Then, the algorithm in equation 11 converges to $\theta^\star$ with probability one.

Proof. First, we can show that $E[\mathbf{Z}_t] = \boldsymbol{G}_t$ by equation 12, the linearity of the derivative operator, and the fact that the permutations are independently sampled for each training example in the mini-batch and are assumed independent of $\boldsymbol{\theta}$. That equation 11 converges to $\boldsymbol{\theta}^\star$ is a consequence of our conditions and the supermartingale convergence theorem (Grimmett & Stirzaker, 2001, pp. 481). The following argument follows Yuille (2004). If $A_t = \frac{1}{2}\|\boldsymbol{\theta}_t - \boldsymbol{\theta}^\star\|_2^2$, $B_t = \frac{1}{2}\delta^2\eta_t^2$, and $C_t = -\|\boldsymbol{\theta}_t - \boldsymbol{\theta}^\star\|_2^2(\delta^2\eta_t^2 - G^2\eta_t)$, noting that C_t is positive for a sufficiently large t, and $\sum_{t=1}^\infty B_t \leq \infty$ by our definition of η_t. The inequality $E[A_t] \leq A_{t-1} + B_{t-1} - C_{t-1}$, for all t, is a consequence of $E\left[\|\boldsymbol{\theta}_t - \boldsymbol{\theta}^\star\|_2^2\right] = E\left[\|\boldsymbol{\theta}_{t-1} - \eta_{t-1}\mathbf{Z}_{t-1} - \boldsymbol{\theta}^\star\|_2^2\right] = E\left[\|\boldsymbol{\theta}_{t-1} - \boldsymbol{\theta}^\star\|_2^2\right] - 2\eta_{t-1}E[(\boldsymbol{\theta}_{t-1} - \boldsymbol{\theta}^\star)^T\mathbf{Z}_{t-1}] + \eta_{t-1}^2 E[\|\mathbf{Z}_{t-1}^2\|_2^2] \leq E\left[\|\boldsymbol{\theta}_{t-1} - \boldsymbol{\theta}^\star\|_2^2\right] - 2\eta_{t-1}(\boldsymbol{\theta}_{t-1} - \boldsymbol{\theta}^\star)^T\boldsymbol{G}_{t-1} + \eta_{t-1}^2 G^2 \leq E\left[\|\boldsymbol{\theta}_{t-1} - \boldsymbol{\theta}^\star\|_2^2\right] - 2\eta_{t-1}G\|\boldsymbol{\theta}_{t-1} - \boldsymbol{\theta}^\star\|_2^2 - \eta_{t-1}(\boldsymbol{\theta}^\star)^T\boldsymbol{G}_t + \eta_{t-1}^2 G^2$. Then, A_t converges to zero with probability one and $\sum_{t=1}^\infty C_t < \infty$. □

There are also ways to cope with the difference in objective functions between equation 9 and equation 12. An interesting approach adds an output regularization penalty for two distinct sampled permutations $\mathbf{s}_i$ and $\mathbf{s}_i'$, $\|\vec{f}(\boldsymbol{h}_{\mathbf{s}_i}(\boldsymbol{x}^{(i)}; \boldsymbol{\theta}^{(h)}); \boldsymbol{\theta}^{(f)}) - \vec{f}(\boldsymbol{h}_{\mathbf{s}_i'}(\boldsymbol{x}^{(i)}; \boldsymbol{\theta}^{(h)}); \boldsymbol{\theta}^{(f)})\|_2^2$, so as to reduce the variance of the sampled Janossy pooling output. Such a variance-reduction procedure allows $E_{\mathbf{s}}[L(y_i, \vec{f}(|\boldsymbol{h}_{\mathbf{s}}|, \boldsymbol{h}_{\mathbf{s}}; \boldsymbol{\theta}^{(f)})] \approx L(y_i, E_{\mathbf{s}}[\vec{f}(|\boldsymbol{h}_{\mathbf{s}}|, \boldsymbol{h}_{\mathbf{s}}; \boldsymbol{\theta}^{(f)})])$, inducing a near-equivalence between optimizing equation 9 and equation 12, assuming ρ is the identity or ρ is combined with $\vec{f}$ to form a single function $\vec{f'} \equiv \rho \circ \vec{f}$. This type of output-regularization-for-variance-reduction approach has been successfully used before to improve test performance of the average Dropout masks on LSTMs (Zolna et al., 2018).

Inference. The use of π-SGD to optimize the Janossy pooling layer optimizes the objective $\overline{\overline{J}}$, and thus has the following implication on how outputs should be calculated at inference time:

Remark 2.2 (Inference)**.** *Assume L is convex (e.g., L is the L2, cross entropy, or negative log-likelihood losses), which via Jensen's inequality makes π-SGD a proper surrogate to the original Janossy objective. At test time we estimate the output $\boldsymbol{y}_i$ of input $\boldsymbol{x}_i$ by computing (or estimating)*

$$\hat{\boldsymbol{y}}_i(\mathbf{x}_i) = E_{\mathbf{s}_i}\left[\vec{f'}(|\boldsymbol{h}^{(i)\star}|, \boldsymbol{h}_{\mathbf{s}_i}^{(i)\star}; \boldsymbol{\theta}^{(f')\star})\right], \tag{13}$$

where $\vec{f'} \equiv \rho \circ \vec{f}$, $\boldsymbol{\theta}^{(f')\star} \equiv (\boldsymbol{\theta}^{(f)\star}, \boldsymbol{\theta}^{(\rho)\star})$, $\boldsymbol{h}_{\mathbf{s}_i}^{(i)\star} \equiv (h(\boldsymbol{x}_i; \boldsymbol{\theta}^{(h)\star}))_{\mathbf{s}_i}$ and $\boldsymbol{\theta}^{(\rho)\star}, \boldsymbol{\theta}^{(f)\star}, \boldsymbol{\theta}^{(h)\star}$ are fixed points of the π-SGD optimization. Equation 13 is a permutation-invariant function.

2.4 Probabilistic Interpretation

Our work has a strong connection with finite exchangeability. Some researchers may be more familiar with the concept of *infinite exchangeability* through de Finetti's theorem (De Finetti, 1937; Diaconis, 1977), which imposes strong structural requirements: the probability of any subsequence must equal the marginalized probability of the original sequence. Finite exchangeability drops this projectivity requirement (Diaconis, 1977), which in general, cannot be simplified beyond first sampling the number of observations m, and then sampling their locations from some exchangeable but non-i.i.d. distribution $p_{\text{xchgx}}^m(x_1, \ldots, x_m)$ (Daley & Vere-Jones, 2003).

Finite exchangeability also arises from the theory of spatial point processes; our framework of Janossy pooling is inspired by *Janossy densities* (Daley & Vere-Jones, 2003), which model the finite exchangeable distributions as mixtures of non-exchangeable distributions applied to permutations. This literature also studies simplied exchangeable point processes such as finite Gibbs models (Vo et al., 2018; Moller & Waagepetersen, 2003) that restrict the structure of p_{xchg} to fixed-order dependencies, and are related to k-ary Janossy.

3 EXPERIMENTS

In what follows we empirically evaluate two tractable Janossy pooling approaches, k-ary dependencies (section 2.2) and sampling permutations for stochastic optimization (section 2.3), to learn permutation-invariant functions for tasks of different complexities. One baseline we compare against is deep sets (Zaheer et al., 2017); recall that this corresponds to unary ($k = 1$) Janossy pooling (Remark 2.1). Corollary 2.1 shows that explicitly modeling higher-order correlations simplifies the task of the upper layers of the neural network, and we evaluate the gains of increasing $k = 1, 2, 3, |\boldsymbol{h}|$ over different tasks. We also evaluate Janossy pooling in graph tasks, where it can be used as a permutation-invariant function to aggregate the features and embeddings of the neighbors of a node in the graph. Note that in graph tasks, permutation-invariance is required to ensure that the neural network is invariant to permutations in the adjacency matrix (graph isomorphism).

3.1 ARITHMETIC TASKS ON SEQUENCES OF INTEGERS

We first consider the task of predicting the *sum* of a sequence of integers (Zaheer et al., 2017), and extend it to consider other harder tasks: *range*, *unique sum*, and *unique count*. In the *sum* task we predict the sum of a sequence of 5 integers drawn uniformly with replacement from $\{0, 1, \ldots, 99\}$; the *range* task also receives a sequence 5 integers distributed the same way, and tries to predict the range (the difference between the maximum and minimum values); the *unique sum* task receives a sequence of 10 integers, sampled uniformly with replacement from $\{0, 1, \ldots, 9\}$, and predicts the sum of all unique elements; the *unique count* task receives a sequence of repeating elements, sampled uniformly with replacement from $\{0, 1, \ldots, 9\}$, and predicts the number of unique elements. Unlike Zaheer et al. (2017), we work with the digits themselves, rather than with MNIST images. This allows a more direct assessment of the different Janossy pooling approximations, and it is a simple matter to pass digit images through another neural network layer before applying our algorithms. Note that sum task maps directly to the architecture of deeps sets (which computes embeddings of the individual digits and adds them together). However the other tasks we selected involved calculating permutation-invariant quantities that require reasoning about high-order relationships within the sequence.

We explore several Janossy pooling variants. First are *Deep Sets, or Janossy ($k = 1$)* and *Janossy ($k = 2, 3$)* where $\overrightarrow{f}$ is a feedforward network with a single hidden layer comprised of 30 neurons. Second, full ($k = |\boldsymbol{h}|$) Janossy pooling where $\overrightarrow{f}$ is an LSTM or a GRU (50 and 80 units in the hidden state, respectively) and the LSTM or GRU hidden state of the last temporal unit is the output of $\overrightarrow{f}$ (the $\boldsymbol{c}_t$ of Cho et al. with $t = |\boldsymbol{h}|$). The full Janossy pooling trained with π-SGD is denoted *Janossy-Minf-LSTM* and *Janossy-Minf-GRU*, where the value of M is the number of sampled permutations used to estimate $\hat{y}_i(\boldsymbol{x}_i)$ of equation 13 at test time for a test sequence $\boldsymbol{x}_i$. To define ρ, we followed the experimental setup of Zaheer et al. (2017), which uses a single dense layer with identity activation. A consequence of this choice is that it provides direct insight into the capacity of $\overline{\overrightarrow{f}}$ to exploit k-wise relationships. We have also tested these tasks where ρ is a feedforward network with one hidden layer using tanh activations and 100 units; some preliminary results from this more powerful nonlinear ρ are also reported.

Table 1 shows the accuracy (average 0-1 loss) for 15 runs of the above Janossy pooling approximations under the *sum*, *range*, *unique sum*, and *unique count* tasks with ρ being a dense layer with identity activation. Table 3 in the Supplementary Material shows the same results measured by mean absolute error. The data consists of 100,000 training examples and 10,000 test examples.
(*Sum* task) Note that the *sum* task (Zaheer et al., 2017) is easy for all Janossy pooling approximations. Also note that for the methods trained with π-SGD (LSTM and GRU), using 20 samples instead of one to estimate equation 13 yields significant accuracy gains.
(*Range*, *unique sum*, and *unique count* tasks) These results corroborate both theorem 2.1 and corollary 2.1, showing that performance increases consistently with k on the tasks that require reasoning about relationships within elements of the sequence. This serves as evidence that optimizing using π-SGD yields strong accuracy gains for modeling the entire sequence, albeit over the modified loss $\overline{\overrightarrow{J}}$ of equation 12. For the methods trained with π-SGD (LSTM and GRU), we also see significant gains of using 20 samples instead of one to estimate equation 13, whenever such gains are possible.

When ρ is a feedforward network with one hidden layer (and 100 hidden units) with tanh activations, our preliminary results (Table 5 in the Supplementary Material) show that Janossy $k = 1$ performs

Table 1: Accuracy (average 0-1 loss) of various Janossy pooling approximations under distinct tasks. Labels -1inf- and -20inf- refer to the number of permutations sampled to estimate equation 13 for methods learned with π-SGD. Standard deviations computed over 15 runs are shown in parentheses.

Tasks	*sum*	*range*	*unique sum*	*unique count*
Janossy (k = 1)-Deep Sets	1.000 (.000)	0.038 (.002)	0.072 (.002)	0.355 (.006)
Janossy (k = 2)	0.999 (.001)	0.090 (.004)	0.173 (.004)	0.693 (.049)
Janossy (k = 3)	0.996 (.001)	0.227 (.004)	0.289 (.017)	0.872 (.015)
Janossy-1inf-LSTM	0.886 (.028)	0.596 (.057)	0.999 (.001)	0.999 (.001)
Janossy-1inf-GRU	0.978 (.027)	0.783 (.024)	1.000 (.000)	1.000 (.000)
Janossy-20inf-LSTM	0.955 (.010)	0.711 (.046)	1.000 (.000)	1.000 (.000)
Janossy-20inf-GRU	0.990 (.001)	0.872 (.017)	1.000 (.000)	1.000 (.000)

Table 2: Performance (Micro-F1) using GraphSAGE with three aggregators shows the benefit of better test-time inference in Janossy pooling approximations trained with π-SGD. Standard deviations over 20 runs in parentheses.

CORA	**CORA**	**PUBMED**
Janossy-1inf-LSTM	0.821 (.005)	0.812 (.007)
Janossy (k = 1)-Deep Sets	0.863 (.004)	.885 (.003)
Janossy-20inf-LSTM	0.857 (.003)	0.889 (.003)

similarly to Table 1 in the *range* task, while Janossy $k = 2, 3$ perform significantly better. For the *unique sum* and unique count tasks Janossy $k = 1, 2, 3$ have significant improvements, while Janossy $k = 2, 3$ either outperforms $k = 1$ or performs equally. These preliminary results also show that Janossy LSTMs and GRUs perform worse when ρ is nonlinear, possibly due to the larger divergence between the objective functions equation 9 and equation 12 when ρ is nonlinear. Investigating this further is a topic for in future work.

3.2 Janossy pooling as an aggregator function for vertex classification

Here we consider the task of vertex classification over two graph datasets: Cora and Pubmed (Sen et al., 2008). These datasets are citation networks where vertices represent papers, edges represent citations, and vertex features are bag-of-words representations of the document text. The task is to classify the paper topic. More details of these datasets and our experimental test/train splits are shown in Table 4 in the Supplementary Material.

We use the graph convolutional model of Hamilton et al. (2017) for its simplicity and the use what the authors call Mean Pooling (Janossy pooling $k = 1$) and *LSTM aggregator*. The *LSTM aggregator* is effectively a Janossy pooling approximation using an LSTM for $\overrightarrow{f}$ and trained with π-SGD, described as Janossy-1inf-LSTM in our results. At test time, however, Hamilton et al. estimates the class label using equation 13 but with a single permutation. Their experiments indicate that mean pooling (Janossy pooling $k = 1$) yields better accuracy than the LSTM approximation on citation network tasks.

Here, we show that gains of mean pooling over the LSTM aggregator reported in Hamilton et al. vanish once we provide a better estimate of equation 13. We note in passing that we have closely followed experiment setup of Hamilton et al. detailed in Section C.1 of the Supplementary Material. The Micro-F1 scores (same metric used by Hamilton et al.) are averaged over 21 runs and shown in Table 2, along with the Janossy $k = 1$ approximation (the mean pooling aggregator of Hamilton et al.). As in (Hamilton et al., 2017), Janossy $k = 1$ seems to have higher accuracy than Janossy-1inf-LSTM (one permutation sample for estimating the node label using equation 13). However, with a label estimate averaged over 20 permutation samples (Janossy-20inf-LSTM), these gains disappear and Janossy-20inf-LSTM accuracy is not significantly different from mean pooling (unary Janossy).

The absence of difference between the unary Janossy and Janossy-LSTM pooling approximations points to an easy task. It would have been pointless to try k-ary Janossy pooling approximations in these tasks. It is reasonable to suppose that the topic of a paper can be adequately predicted by computing the average bag-of-words representations of papers in the neighborhood without reasoning about relationships between neighboring papers (i.e. it may be a small-k task). For example, a high proportion of mentions of *neural* in neighboring papers is suggestive about the paper topic and can be computed independently for each paper.

4 RELATED WORK

Under the Janossy pooling framework presented in this work, existing literature falls under one of three approaches to approximating to the intractable Janossy-pooling layer: *Canonical orderings*, *k-ary dependencies*, and *permutation sampling*.

Canonical Ordering Approaches. In section 2.1, we saw how permutation invariance can be achieved by mapping permutations to a canonical ordering. Rather than trying to define a good canonical ordering, one can try to learn it from the data, however searching among all $\boldsymbol{h}!|$ permutations for one that correlates with the task of interest is a difficult discrete optimization problem. Recently, Rezatofighi et al. (2018) proposed a method that computes the posterior distribution of all permutations, conditioned on the model and the data. This posterior-sampling approach is intractable for large inputs, unfortunately. We note in passing that Rezatofighi et al. (2018) is interested in permutation-invariant outputs, and that Janossy pooling is also trivially applicable to these tasks. Vinyals et al. (2016) proposes a heuristic using ancestral sampling while learning the model.

k-ary Janossy Pooling Approaches. In section 2.2 we described k-ary Janossy pooling, which considers k-order relationships in the input vector $\boldsymbol{h}$ to simplify optimization. Deep Sets (Zaheer et al., 2017) can be characterized as unary Janossy pooling (i.e., k-ary for $k = 1$). Our experiments considered tasks that see significant gains from higher-order k-ary Janossy pooling, $k > 1$. Ravanbakhsh et al. (2017) proposes a similar unary Janossy pooling to *deep sets*. Cotter et al. (2018) proposes to add inductive biases to the *deep sets* model in the form of monotonicity constraints with respect to the vector valued elements of the input sequence by modeling ϕ and ρ with Deep Lattice Networks (You et al., 2017); one can extend Cotter et al. (2018) by using higher-order k-ary Janossy pooling for $k > 1$.

Exploiting dependencies within a sequence to learn a permutation invariant function has been discussed elsewhere. For instance Santoro et al. (2017) exploits pairwise relationships to perform relational reasoning about pairs of objects in an image Battaglia et al. (2018) contemplates modeling the center of mass of a solar system by including the pairwise interactions among planets. However, Janossy pooling provides a general framework for capturing dependencies within a permutation invariant pooling layer.

Permutation Sampling Approaches. In section 2.3 we have seen a that permutation sampling can be used as a stochastic gradient procedure (π-SGD) to learn a model with a Janossy pooling layer. The learned model provides only an approximate solution to original permutation-invariant function. Permutation sampling has been used as a heuristic (without a theoretical justification) in both Moore & Neville (2017) and Hamilton et al. (2017), which found that randomly permuting sequences and feeding them forward to an LSTM is effective in relational learning tasks that require permutation invariant pooling layers.

5 CONCLUSIONS

Our approach of permutation-invariance through Janossy pooling unifies a number of existing approaches, and opens up avenues to develop both new methodological extensions, as well as better theory. Our paper focused on two main approaches: k-ary interactions and random permutations. The former involves exact Janossy pooling for a restricted class of functions $\vec{f}$. Adding an additional neural network ρ can recover lost model capacity, but now hurts learnability. Placing restrictions on ρ (convexity, Lipschitz continuity etc.) can allow a more refined control of this trade-off, allowing theoretical and empirical work to shed light on the compromises involved. The random permutation approach involves no clear trade-offs between model capacity, and learnability, instead it approximates the Janossy loss $\overline{\overline{L}}$ with a more tractable approximation $\overline{\overline{J}}$. The difference between the two reduces as more permutations are sampled, however, despite excellent empirical performance, it is not clear what problems this is best suited for. Better understanding how the loss-functions $\overline{\overline{L}}$ and $\overline{\overline{J}}$ relate to each other, can shed light on the slightly black-box nature of this procedure. It is also important to understanding the relationship between the random-permutation approach to canonical ordering, and how one might be used to improve the other. Finally, it is important to apply our methodology to a wider range of applications. Two immediate domains are more challenging tasks involving graphs and tasks involving non-Poisson point processes.

REFERENCES

James Atwood and Don Towsley. Diffusion-convolutional neural networks. In *NIPS*, 2016.

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.

Léon Bottou and Yann L Cun. Large scale online learning. In *NIPS*, 2004.

Kyunghyun Cho, Bart Van Merriënboer, Caglar Gulcehre, Dzmitry Bahdanau, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. Learning phrase representations using rnn encoder-decoder for statistical machine translation. *EMNLP*, 2014.

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

Andrew Cotter, Maya Gupta, Heinrich Jiang, James Muller, Taman Narayan, Serena Wang, and Tao Zhu. Interpretable set functions. *arXiv preprint arXiv:1806.00050*, 2018.

Daryl J Daley and David Vere-Jones. *An introduction to the theory of point processes: volume II: general theory and structure*. Springer, 2003.

Bruno De Finetti. La prévision: ses lois logiques, ses sources subjectives. In *Annales de l'institut Henri Poincaré*, volume 7, pp. 1–68, 1937. [Translated into Enlish: H. E. Kyburg and H.E. Smokler, eds. Studies in Subjective Probability. *Krieger* 53-118, 1980].

Persi Diaconis. Finite forms of de Finetti's theorem on exchangeability. *Synthese*, 36(2):271–281, 1977.

Persi Diaconis and David Freedman. De finetti's generalizations of exchangeability. *Studies in inductive logic and probability*, 2:233–249, 1980.

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. In *NIPS*, 2015.

Morris L. Easton. *Chapter 8: Finite de Finetti style theorems*, volume Volume 1 of *Regional Conference Series in Probability and Statistics*, pp. 108–120. Institute of Mathematical Statistics and American Statistical Association, Haywood CA and Alexandria VA, 1989. URL `https://projecteuclid.org/euclid.cbms/1462061038`.

Andrew Gelman, John B. Carlin, Hal S. Stern, David B. Dunson, Aki Vehtari, and Donald B. Rubin. *Bayesian Data Analysis*. CRC Press, 3 edition, 2014.

Justin Gilmer, Samuel S. Schoenholz, Patrick F. Riley, Oriol Vinyals, and George E. Dahl. Neural message passing for quantum chemistry. In *ICML*, 2017.

Geoffrey Grimmett and David Stirzaker. *Probability and random processes*. Oxford university press, 2001.

William L. Hamilton, Rex Ying, and Jure Leskovec. Inductive Representation Learning on Large Graphs. In *NIPS*, jun 2017. URL `http://arxiv.org/abs/1706.02216`.

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

Diederik P. Kingma and Jimmy Lei Ba. ADAM: A Method for Stochastic Optimization. *International Conference on Learning Representations, ICLR*, 2015.

Thomas N. Kipf and Max Welling. Semi-Supervised Classification with Graph Convolutional Networks. sep 2016. URL `http://arxiv.org/abs/1609.02907`.

Yann LeCun, Yoshua Bengio, et al. Convolutional networks for images, speech, and time series. *The handbook of brain theory and neural networks*, 3361(10):1995, 1995.

Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. Deep learning. *Nature*, 2015.

David Liben-Nowell and Jon Kleinberg. The link-prediction problem for social networks. *Journal of the American society for information science and technology*, 58(7):1019–1031, 2007.

Ziqi Liu, Chaochao Chen, Longfei Li, Jun Zhou, Xiaolong Li, and Le Song. Geniepath: Graph neural networks with adaptive receptive paths. *arXiv preprint arXiv:1802.00910*, 2018.

Jesper Moller and Rasmus Plenge Waagepetersen. *Statistical inference and simulation for spatial point processes*. Chapman and Hall/CRC, 2003.

Federico Monti, Davide Boscaini, Jonathan Masci, Emanuele Rodola, Jan Svoboda, and Michael M Bronstein. Geometric deep learning on graphs and manifolds using mixture model cnns. In *Proc. CVPR*, volume 1, pp. 3, 2017.

John Moore and Jennifer Neville. Deep collective inference. In *AAAI*, pp. 2364–2372, 2017.

Mathias Niepert, Mohamed Ahmed, and Konstantin Kutzkov. Learning convolutional neural networks for graphs. In *International conference on machine learning*, pp. 2014–2023, 2016.

Peter Orbanz and Daniel M. Roy. Bayesian Models of Graphs, Arrays and Other Exchangeable Random Structures. *IEEE Trans. Pattern Anal. Mach. Intell.*, 37(2):437–461, feb 2015. ISSN 0162-8828. doi: 10.1109/TPAMI.2014.2334607. URL `http://ieeexplore.ieee.org/document/6847223/`.

Siamak Ravanbakhsh, Jeff Schneider, and Barnabas Poczos. Deep Learning with Sets and Point Clouds. In *ICLR Workshop Track*, nov 2017. URL `http://arxiv.org/abs/1611.04500`.

S Hamid Rezatofighi, Roman Kaskman, Farbod T Motlagh, Qinfeng Shi, Daniel Cremers, Laura Leal-Taixé, and Ian Reid. Deep perm-set net: Learn to predict sets with unknown permutation and cardinality using deep neural networks. *arXiv preprint arXiv:1805.00613*, 2018.

H. Robbins and S. Monro. A stochastic approximation method. *Annals of Mathematical Statistics*, 1951.

Adam Santoro, David Raposo, David G Barrett, Mateusz Malinowski, Razvan Pascanu, Peter Battaglia, and Tim Lillicrap. A simple neural network module for relational reasoning. In *Advances in neural information processing systems*, pp. 4967–4976, 2017.

Prithviraj Sen, Galileo Namata, Mustafa Bilgic, Lise Getoor, Brian Galligher, and Tina Eliassi-Rad. Collective classification in network data. *AI magazine*, 29(3):93, 2008.

Rianne van den Berg, Thomas N Kipf, and Max Welling. Graph convolutional matrix completion. *stat*, 1050:7, 2017.

Petar Velickovic, Guillem Cucurull, Arantxa Casanova, Adriana Romero, Pietro Lio, and Yoshua Bengio. Graph attention networks. *arXiv preprint arXiv:1710.10903*, 2017.

Oriol Vinyals, Samy Bengio, and Manjunath Kudlur. Order Matters: Sequence to Sequence for Sets. *ICLR*, 2016.

Ba-Ngu Vo, Nhan Dam, Dinh Phung, Quang N Tran, and Ba-Tuong Vo. Model-based learning for point pattern data. *Pattern Recognition*, 84:136–151, 2018.

Keyulu Xu, Chengtao Li, Yonglong Tian, Tomohiro Sonobe, Ken-ichi Kawarabayashi, and Stefanie Jegelka. Representation Learning on Graphs with Jumping Knowledge Networks. In *ICML*, 2018. URL `http://arxiv.org/abs/1806.03536`.

Rex Ying, Jiaxuan You, Christopher Morris, Xiang Ren, William L Hamilton, and Jure Leskovec. Hierarchical graph representation learning with differentiable pooling. *arXiv preprint arXiv:1806.08804*, 2018.

Seungil You, David Ding, Kevin Canini, Jan Pfeifer, and Maya Gupta. Deep lattice networks and partial monotonic functions. In *Advances in Neural Information Processing Systems*, pp. 2981–2989, 2017.

Alan Yuille. The Convergence of Contrastive Divergences. In *NIPS*, 2004.

Manzil Zaheer, Satwik Kottur, Siamak Ravanbakhsh, Barnabas Poczos, Ruslan Salakhutdinov, and Alexander Smola. Deep Sets. In *NIPS*, 2017.

Konrad Zolna, Devansh Arpit, Dendi Suhubdy, and Yoshua Bengio. Fraternal dropout. *ICLR*, 2018.

A PROOFS OF RESULTS

We restate and prove the remaining portion of theorem 2.1.

Theorem 2.1. *For any $k \in \mathbb{N}$, define $\mathcal{F}_k$ as the set of all permutation invariant functions that can be represented by Janossy pooling with k-ary dependencies. Then, $\mathcal{F}_{k-1}$ is a* proper *subset of $\mathcal{F}_k$ if the space $\mathbb{H}$ is not trivial (i.e. if the cardinality of $\mathbb{H}$ is greater than 1). Thus, Janossy pooling with k-ary dependencies can express any Janossy pooling function with $(k-1)$-ary dependencies, but the converse does not hold.*

Proof. ($\mathcal{F}_k \not\subset \mathcal{F}_{k_1}$):

We will demonstrate the existence of $\overline{\overline{f}}_k \in \mathcal{F}_k$ such that $\overline{\overline{f}}_{k-1} \neq \overline{\overline{f}}_k$ for all $\overline{\overline{f}}_{k-1} \in \mathcal{F}_{k-1}$. It suffices to consider $|\boldsymbol{x}| = k$. Let $\vec{f}_k(|\boldsymbol{x}|, \boldsymbol{x}_\pi; \boldsymbol{\theta}) = \prod_{l=1}^{|\boldsymbol{x}|} x_{\pi(l)}$ whence $\overline{\overline{f}}_k(\boldsymbol{x}_\pi; \boldsymbol{\theta}) = \prod_{l=1}^{|\boldsymbol{x}|} x_l$. Thus, for any $\overline{\overline{f}}_{k-1}$,

$$\frac{\overline{\overline{f}}_{k-1}(|\boldsymbol{x}|, \boldsymbol{x}_\pi; \boldsymbol{\theta})}{\overline{\overline{f}}_k(|\boldsymbol{x}|, \boldsymbol{x}_\pi; \boldsymbol{\theta})} = \frac{1}{|\boldsymbol{x}|!} \sum_{\pi \in \Pi_{|\boldsymbol{x}|}} \frac{\vec{f}_{k-1}(k-1, \downarrow_{k-1}(\boldsymbol{x}_\pi); \boldsymbol{\theta})}{\prod_{l=1}^{|\boldsymbol{x}|} x_l}$$

$$= \frac{1}{|\boldsymbol{x}|!} \sum_{j=1}^{|\boldsymbol{x}|} \sum_{\tilde{\pi} \in \Pi_{\{1,\ldots,|\boldsymbol{x}|\} \setminus j}} \frac{\vec{f}_{k-1}\big(k-1, (\boldsymbol{x}_{-j})_{\tilde{\pi}}; \boldsymbol{\theta}\big)}{\prod_{l=1}^{|\boldsymbol{x}|} x_l}$$

where $\Pi_{\{1,\ldots,|\boldsymbol{x}|\} \setminus j}$ is the set of permutation functions of $\{1, 2, \ldots, j-1, j+1, \ldots, |\boldsymbol{x}|\}$ and $(\boldsymbol{x}_{-j})_{\tilde{\pi}}$ is a permutation of $\{x_1, \ldots, x_{j-1}, x_{j+1}, \ldots, x_{|\boldsymbol{x}|}\}$.This can be written as

$$\frac{1}{|\boldsymbol{x}|!} \sum_{j=1}^{|\boldsymbol{x}|} \frac{1}{x_j} \underbrace{\Big(\sum_{\tilde{\pi} \in \Pi_{\{1,\ldots,|\boldsymbol{x}|\} \setminus j}} \frac{\vec{f}_{k-1}\big(k-1, (\boldsymbol{x}_{-j})_{\tilde{\pi}}; \boldsymbol{\theta}\big)}{\prod_{l \neq j} x_l} \Big)}_{\text{denote by } a_{j,|\boldsymbol{x}|}},$$

whence

$$\frac{\overline{\overline{f}}_{k-1}(|\boldsymbol{x}|, \boldsymbol{x}_\pi; \boldsymbol{\theta})}{\overline{\overline{f}}_k(|\boldsymbol{x}|, \boldsymbol{x}_\pi; \boldsymbol{\theta})} = \frac{1}{|\boldsymbol{x}|!} \sum_{j=1}^{|\boldsymbol{x}|} \frac{1}{x_j} a_{j,|\boldsymbol{x}|}. \tag{14}$$

Now, $\overline{\overline{f}}_{k-1} = \overline{\overline{f}}_k$ if and only if their quotient in equation 14 is unity for all $\boldsymbol{x}$. But this is clearly not possible in general unless $\mathbb{X}$ is a singleton, which we have precluded in our assumptions. □

B DE FINETTI'S THEOREMS

The infinite form of de Finetti's theorem arises when contemplating an infinite sequence of exchangeable random variables or when contemplating a finite sequence whose distribution could arise as a marginalization of an infinitely long exchangeable sequence (Diaconis, 1977; Orbanz & Roy, 2015; Gelman et al., 2014). When this holds, the data distribution $p(X_1, X_2, \ldots)$ can be represented as a mixture of some prior distribution $p(\theta)$ for parameter $\theta \in \Theta$ and a conditionally iid likelihood for the data, given θ: in the case of an infinite exchangeable sequence[1],

$$p(X_1, X_2, \ldots,) = \int_{\theta \in \Theta} \prod_{i=1}^{\infty} p(X_i \mid \theta) p(\theta) d\theta \tag{15}$$

[1]Here we write a slightly simplified form, as in Gelman et al. (2014) which is satisfactory for our discussion here. A more general form can be found in Orbanz & Roy (2015).

Table 3: Mean Absolute Error of various Janossy pooling across various tasks. Labels -1inf- and -20inf- refer to the number of permutations sampled to estimate equation 13 for methods learned with π-SGD. Standard deviations computed over 15 runs are shown in parentheses.

Tasks	*sum*	*range*	*unique sum*	*unique count*
Janossy-MLP ($\mathbf{k=1}$)-Deep Sets	0.026(.005)	9.424(.081)	4.235(.020)	0.828(.008)
Janossy-MLP ($\mathbf{k=2}$)	0.032(.007)	4.057(.042)	1.972(.023)	0.393(.006)
Janossy-MLP ($\mathbf{k=3}$)	0.032(.003)	2.265(.051)	1.157(.088)	0.266(.010)
Janossy-1inf-LSTM	0.510(.055)	0.630(.101)	0.029(.008)	0.017(.005)
Janossy-1inf-GRU	0.263(.069)	0.364(.028)	0.01(.001)	0.005(.001)
Janossy-20inf-LSTM	0.433(.040)	0.482(.079)	0.014(.006)	0.009(.003)
Janossy-20inf-GRU	0.194(.011)	0.249(.022)	0.005(.001)	0.003(0)

or if $(X_1, \ldots, X_n)$ can be extended to an infinitely exchangeable sequence, its distribution is arbitrarily close to the form in equation 15Diaconis (1977).

However, Diaconis (1977) shows that not all finitely exchangeable sequences can be written as in equation 15, by demonstrating that one can write an exchangeable probability mass function of two binary random variables (over events $\{0,0\}, \{0,1\}, \{1,0\}, \{1,1\}$) where $p(\{0,1\}) = p(\{1,0\})$, which cannot be written in general as a mixture over conditionally iid Bernoulli random variables.

Conversely, a representational form for exchangeable *finite* sequences has also been studiedDiaconis & Freedman (1980); Easton (1989). For example, in the case of m binary random variables $X_1, X_2, \ldots, X_m$, the probability distribution is represented by the following generating process: (1) randomly select the number of successes $\sum_{i=1}^{m} X_i = s < m$ in an urn with m balls , then (2) randomly draw balls from that urn. Observe that if an urn has s successes, and the first s draws are all successes, the $(s+1)^{\text{th}}$ draw must be a failure, and thus the draws are not iid.

C Experiments: Further Results and Implementation Details

C.1 Results

Mean Absolute errors for arthimetic tasks on integer sequences are reported in Table 3. These largely corroborate the results from Table 1, with a drop in the mean absolute error as the value of k increases and when using more sampled permutations at test-time (e.g., Janossy-20inf-LSTM versus Janossy-1inf-LSTM). Beyond the performance gains, we also observe a drop in variance when sampling more permutations at test time.

C.2 Implementation Details

Sequence tasks We extended the code from Zaheer et al. (2017), which was written in Keras(Chollet et al., 2015). Regardless of the model, we always sort the sequence $\boldsymbol{x}$ beforehand. In the notation of Figure 1, $\boldsymbol{h}$ is an Embedding with dimension of $\mathrm{floor}(\frac{100}{k})$, $\vec{f}$ is either an MLP with a single hidden layer or an RNN depending on the model (k-ary Janossy or full-Janossy, respectively), and ρ is always a single dense layer. The MLPs have 30 neurons, the LSTMs have 50 neurons, and the GRUs have 80 hidden neurons. All activations are $\tanh$ except for the output layer which is linear.

We constructed these models to be as comparable as possible; each has a similar number of total parameters and the same number of layers. We unify the number of parameters by adjusting the output dimension of the embedding (approximately $\frac{1}{k}$) for the different k-ary models and by using fewer neurons in the LSTM than the GRU since LSTMs are more complicated models with more parameters.

Optimization is done with Adam (Kingma & Ba, 2015) using a learning rate of 10^{-4}. We did not do a tuning parameter sweep. Training was performed on GeForce GTX 1080 Ti GPUs.

Table 4: Summary of the graph datasets

CHARACTERISTICS	CORA	PUBMED
Number of Vertices	2708	19717
Average Degree	3.900	4.496
Number of Vertex Features	1433	500
Number of Classes	7	3
Number of Training Vertices	256	256
Number of Test Vertices	1000	1000

Table 5: Preliminary results similar to Table 1: accuracy of various Janossy pooling models across various tasks where ρ is an MLP

Tasks	*range*	*unique sum*	*unique count*
Janossy-MLP ($\mathbf{k=1}$)-Deep Sets	0.049 (.002)	0.361 (.040)	1 (0)
Janossy-MLP ($\mathbf{k=2}$)	0.379 (.023)	0.970 (.015)	1 (0)
Janossy-MLP ($\mathbf{k=3}$)	0.616 (.022)	0.918 (.024)	1 (0)
Janossy-1inf-LSTM	0.226 (.014)	0.693 (.042)	0.631(.073)
Janossy-1inf-GRU	0.188 (.015)	0.925 (.024)	0.998 (.003)
Janossy-20inf-LSTM	0.238 (.020)	0.669 (.202)	0.626 (.116)
Janossy-20inf-GRU	0.199 (.028)	0.937 (.052)	0.995 (.010)

Graph-based tasks The characteristics of the datasets used are tabulated in Table 4. This task is implemented in PyTorch and trained on the same GeForce GTX 1080 Ti GPUs.

As this experiment is using Janossy pooling as a GraphSAGE (Hamilton et al., 2017) aggregator, we forked the GraphSAGE datasets and code for experimental set-up and the mean-pool aggregator; we implemented our own LSTM as it is not yet available in the GraphSAGE PyTorch repo. Following Hamilton et al. (2017) a depth (i.e. number of aggregator steps) of 2, an Adam optimizer and a hidden LSTM dimension of 256. The remaining hyperparameters are dataset-specific:

- On the Cora dataset, for all models, the learning rate is 0.005; the number of random samples at depth one and two are both five.
- On the Pubmed dataset, for all models the learning rate is 0.01; the number of random samples at depth one and two are 10 and 25, respectively.

Note that the embedding function h, in terms of Figure 1 is an identity in this task.

D Supplementary results on Sequence Arithmetic Tasks

Here we present our preliminary results on the sequence task where ρ is modeled as a feedforward network with one hidden layer containing 100 hidden neurons and $\tanh$ activations. Other than the model for ρ, the implementation details are as in section C.1. No hyperparameter tuning was performed. We are not considering results for the sum task since the experimental interest is to investigate whether a richer model for ρ can carry the burden of exploiting dependencies in the sequence when $k = 1$. The results are shown in Table 5. We can see that deep sets does not perform well across tasks and that increasing k helps to boost performance.