HATS: A Hierarchical Sequence-Attention Framework for Inductive Set-of-Sets Embeddings

Changping Meng[1], Jiasen Yang[2], Bruno Ribeiro[1], Jennifer Neville[1,2]
[1]Department of Computer Science, [2]Department of Statistics
Purdue University
West Lafayette, IN
{meng40,jiaseny,ribeirob,neville}@purdue.edu

ABSTRACT

In many complex domains, the input data are often not suited for the typical vector representations used in deep learning models. For example, in relational learning and computer vision tasks, the data are often better represented as *sets* (*e.g.*, the neighborhood of a node, a cloud of points). In these cases, a key challenge is to learn an embedding function that is *invariant* to permutations of the input. While there has been some recent work on principled methods for learning permutation-invariant representations of *sets*, these approaches are limited in their applicability to *set-of-sets* (SoS) tasks, such as subgraph prediction and scene classification. In this work, we develop a deep neural network framework to learn inductive SoS embeddings that are invariant to SoS permutations. Specifically, we propose HATS, a hierarchical sequence model with attention mechanisms for inductive set-of-sets embeddings. We develop stochastic optimization and inference methods for learning HATS, and our experiments demonstrate that HATS achieves superior performance across a wide range of set-of-sets tasks.

KEYWORDS

Set-of-sets; permutation-invariance; inductive embedding; sequence model; long-short term memory network; hierarchical attention.

ACM Reference Format:
Changping Meng, Jiasen Yang, Bruno Ribeiro, Jennifer Neville. 2019. HATS: A Hierarchical Sequence-Attention Framework for Inductive Set-of-Sets Embeddings. In *The 25th ACM SIGKDD Conference on Knowledge Discovery and Data Mining (KDD ’19), August 4–8, 2019, Anchorage, AK, USA.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3292500.3330876

1 INTRODUCTION

Deep learning has been successfully applied to numerous applications in which the input data typically involves fixed-length vectors. Examples include image recognition, video classification, sentiment analysis, among many others. A critical aspect of vector representations is that the position of elements matter.

In many complex tasks, however, the input data are not well-suited for vector representations. In particular, some domains exhibit heterogeneous structure and the data are often better represented as *sets*: *e.g.*, relational learning [12, 23, 24, 26, 30, 37], logical reasoning [18, 31], scene understanding [3, 30], and object detection from LiDAR readings [27, 40]. As there is no natural ordering to the elements in a set, any model must jointly learn functions over all the set elements in order to capture relational dependencies.

Initial work on learning neural-network models for heterogeneous set-inputs often transformed the data into variable-length sequences [11, 12, 24]. However, these methods learn models (*i.e.*, *embedding functions*) that are *permutation-sensitive*. In other words, the output of the learned model (*i.e.*, embedding) depends on the order chosen for the input vector. For example, recurrent neural networks, which are commonly used in sequence-to-sequence (*seq2seq*) models, are not invariant to permutations in the input sequence. Indeed, Vinyals et al. [35] showed that the order of the input sequence could significantly affect the quality of the learned model.

A more principled approach to learning functions over sets is to learn an *inductive* embedding function that is *permutation invariant* and which could be used to directly model the sets themselves. More recent work has focused on developing principled approaches to learning these set representations [19, 25, 35, 39, 40]. The key contribution of these works has been to provide scalable methods that can learn inductive embeddings which are provably invariant to permutations of the input.

While these recent works provide principled approaches to learning over sets, they are not directly applicable to tasks where each data instance is a *set-of-sets* (SoS). For example, subgraph prediction tasks in relational data involve a *set* of nodes, each of which has a *set* of neighbors. In LiDAR scene classification, each scene consists of a *set* of *point-clouds*. Set-of-sets also arise in logical reasoning, multi-instance learning, among other applications (see Section 5 for a few examples). In these tasks, the embedding function to be learned needs to be invariant to two levels of permutations on the input data—within each set, and among the sets. Effective neural network architectures that learn inductive SoS embeddings need to efficiently take into account both levels of permutation-invariance.

In this work, we formalize the problem of learning *inductive embedding* functions over SoS inputs, and explore neural network architectures for learning inductive set-of-sets embeddings. We shall use the terms *set* and *multisets* (sets with repeated elements) interchangeably, as our techniques apply to both scenarios.

Contributions. We begin by proving that inductive embeddings for sets [25, 40] are not expressive enough to be used for SoS tasks. We then propose a general framework for learning inductive SoS embeddings that preserve SoS permutation-invariance by extending the characterization of sets in Murphy et al. [25] to sets-of-sets.

Our framework allows us to apply sequence models to learning inductive SoS embeddings. We propose HATS, a *h*ierarchical, sequence-*at*tention architecture for learning inductive *s*et-of-sets embeddings. HATS utilizes two bidirectional long short-term memory (LSTM) networks. The first LSTM takes as input a sequence of elements arranged in random order. This LSTM is applied to each member set of the SoS, and the outputs are collectively fed into a second bidirectional LSTM. The HATS architecture uses the concept of *hierarchical attention* for sequence models, first introduced in the context of document classification by Yang et al. [38]. However, unlike the model in [38] (which is permutation-sensitive), HATS is based on our SoS embedding framework and uses *permutation sampling* to perform stochastic optimization. This allows HATS to learn inductive SoS embeddings that are provably invariant to SoS permutations. At inference time, we adopt an efficient Monte Carlo procedure that approximately preserves this invariance.

In experiments, we show that HATS significantly outperforms existing approaches on various predictive tasks involving SoS inputs. While the SoS permutation-invariance of HATS is only approximate due to our reliance on Monte Carlo sampling for inference, our experiment results demonstrate that in practice HATS achieves significant performance-gains over state-of-the-art approaches.

2 RELATED WORK

Learning inductive set-embeddings has attracted much attention in the literature [6, 8, 15, 19, 25, 29, 35, 39, 40]. In particular, Zaheer et al. [40] provided a characterization for permutation-invariant functions on sets, and designed a deep neural network architecture, called *deep sets*, that could operate on set inputs. This approach was later generalized by Murphy et al. [25], who proposed a general framework for pooling operations on set inputs, termed *Janossy pooling*. The key observation underlying their approach is that any permutation-invariant function could be expressed as the average of a permutation-sensitive function applied to all re-orderings of the input sequence. Our characterization of sets-of-sets functions in Section 3.2 is directly motivated by this insight.

Among other works, Ilse et al. [15] examined the task of *multi-instance learning* [10, 22], where a class label is assigned to a set of instances (called a *bag*), rather than a single instance. They proposed to learn a permutation-invariant aggregation function using a combination of convolutional neural networks (CNNs) and fully connected layers as well as gated attention mechanisms. Vinyals et al. [35] examined sequence-to-sequence (seq2seq) models with set inputs and showed that standard seq2seq models such as LSTMs are not invariant to permutations in the input; Lee et al. [19] proposed attention-based architectures building upon transformer networks [34]; Bloem-Reddy and Teh [6] studied the group invariance properties of neural network architectures from the perspective of probabilistic symmetries.

The primary distinction between this work and previous works on learning set functions lies in our observation that the input examples in many applications are in fact *sets-of-sets*, rather than (plain) sets, and we argue that their hierarchical nature deserves particular treatment in modeling. This motivates us to move beyond (single-level) sequence models to hierarchical ones when designing the neural network architecture (see Section 4 for details), and our experiments demonstrate that the hierarchical models yield significant performance-gains in practice. We note that Hartford et al. [13] studied matrix-factorization models for learning interactions across two sets (users and movies) in the context of recommender systems. However, their Kronecker-product–based approach is transductive rather than inductive, and is designed for a very specific application, whereas we are interested in general inductive embedding approaches for sets-of-sets.

As with most recent works in the literature, we choose to parametrize the set-of-sets (SoS) permutation-invariant function using deep neural networks, thanks to their expressiveness as universal function approximators. Regarding the choice of neural network architectures, we focus on recurrent neural networks (RNNs)—in particular, long short-term memory (LSTM) networks. The choice is in contrast to that made by Ilse et al. [15], who focused on convolution-based approaches. As with many other works in the literature [25, 35], we believe that sequence models are more appropriate for modeling variable-size inputs. In Section 5 we demonstrate empirically that the proposed LSTM-based models lead to improved performance over the CNN-based model of [15] across a variety of tasks and the less specialized use of LSTMs in [25]. We further investigate attention-based mechanisms for hierarchical LSTM models to enhance their capability of capturing long-range dependencies. We note that similar hierarchical-attention architectures have been considered for document classification [38] in the natural language processing literature, but we adapt it to modeling SoS functions in order to preserve permutation-invariance.

Methods for learning permutation-invariant functions on set structures have direct implications to relational learning and graph mining [7, 12, 24, 37], point-cloud modeling [27, 28] and scene understanding [3, 30] in computer vision, among other applications. While we have conducted experiments on subgraph hyperlink prediction and point-cloud classification tasks to evaluate the performance of our proposed approaches, we emphasize that (as in *e.g.*, [15, 25, 40]) the aim of our work is to provide a general characterization and framework for modeling functions with sets-of-sets inputs, rather than outperforming state-of-the-art approaches that are crafted for specific applications. Importantly, the generality of our proposed approach provides practitioners with the flexibility of tailoring it to their specific tasks at hand.

3 INDUCTIVE SET-OF-SETS EMBEDDINGS

In this section, we provide a formal definition of inductive SoS embeddings. An *inductive embedding* is a function that takes any set-of-sets as input (including sets-of-sets not observed in the training data) and outputs an embedding that must remain unchanged for any input that represents the same set-of-sets. This is in contrast to transductive embedding methods such as matrix and tensor factorizations, which cannot be directly applied to new data and do not directly consider set-of-sets inputs. For convenience, henceforth we shall use the term *sets* to refer to both sets and *multisets*. Multisets are sets that allow duplicate elements [5].

3.1 Set-of-Sets (SoS) Inputs

A *set-of-sets* (SoS) is a multiset whose elements are themselves multisets (which will shall refer to as *member sets*). The multisets belonging to an SoS can have varying sizes. An SoS can be naturally represented as a *list* of *lists* of vectors; in this work, we shall represent an SoS as an $n \times m \times d$ tensor $\mathbf{A}$, whose i-th "row" slice $\mathbf{A}_{i,*}$ corresponds to the i-th multiset, and the (i, j)-th fiber A_{ij} is a d-dimensional vector corresponding to the j-th element of the i-th multiset.[1] Here, n is the number of member sets in the SoS, and m equals the cardinality of the largest member set (for the other multisets, we pad their columns with a special *null* symbol "#"). As an example, a point-cloud *scene* (which we shall examine in Section 5) is a set of point-clouds, and each point is a vector in $\mathbb{R}^3$. In this case, $\mathbf{A}$ is a three-mode tensor, with each fiber $A_{ij} \in \mathbb{R}^3$.

For an $n \times m \times d$ tensor $\mathbf{A}$ representing an SoS, the first two modes of $\mathbf{A}$ must be treated as sets, while the third mode is treated as a vector. The key distinction between a *set* and a *vector* is that (i) a set may contain an arbitrary number of elements, while a vector is generally of fixed-length; and more importantly, (ii) a set is invariant to arbitrary permutations of its elements: if one is to arrange the elements of a set in a vector $\boldsymbol{a} = [a_i]_{i=1}^m$, then for any permutation $\pi : i \mapsto \pi(i)$ on the integers $\{1, \ldots, m\}$, both $\boldsymbol{a}$ and $\boldsymbol{a}_\pi$ represents the same set (here, $\boldsymbol{a}_\pi$ denotes the permuted vector with $(\boldsymbol{a}_\pi)_i = a_{\pi(i)}$). This means that if a function f is a representation of a set, then $f(\boldsymbol{a}) = f(\boldsymbol{a}_\pi)$.

In this work, we shall be interested in functions that yield inductive embeddings for set-of-sets inputs. Such SoS embeddings should satisfy permutation-invariance on two different levels: (i) *element-level*: the function should be invariant to permutations of elements within each member set; and (ii) *member-set-level*: the function should be invariant to permutations of the member sets within an SoS. We formalize these notions in the next section.

3.2 Inductive SoS Embeddings

Notation. Let Π_n denote the set of all permutations on the integers $\{1, \ldots, n\}$. We shall adapt the notation in Murphy et al. [25] and use a double-bar (as in $\overline{\overline{f}}$) to indicate that a function taking SoS inputs is invariant to permutations in the sense of Definition 1 (below). We shall use an arrow (as in $\vec{f}$) to denote arbitrary (possibly permutation-sensitive) functions over SoS inputs. Functions over scalars or vectors will be denoted without such annotations.

3.2.1 *Definition and characterization.*

We begin by defining functions that yield inductive set-of-sets (SoS) embeddings.

Definition 1 (Inductive SoS Embedding). *A function $\overline{\overline{f}}$ acting on SoS inputs is an* inductive SoS embedding *if its output is invariant under any permutation ϕ of the member sets, as well as permutations $\pi_1, \ldots, \pi_n$ of the elements in each member set. Formally, for any $n \times m \times d$ tensor $\mathbf{A}$,*

$$\overline{\overline{f}}(\mathbf{A}) = \overline{\overline{f}}(\mathbf{A}_{\phi,\pi_\phi}), \quad (1)$$

where $\mathbf{A}_{\phi,\pi_\phi}$ denotes the $n \times m \times d$ tensor with (i, j)-th fiber $A_{i,j}$ equal to $A_{\phi(i),\pi_{\phi(i)}(j)}$. Equation (1) should hold for any permutation ϕ of the integers $\{1, \ldots, n\}$, as well as permutations $\{\pi_i\}_{i=1}^n$ of $\{1, \ldots, m\}$.[2]

[1] In tensor terminology, the *mode* of a tensor is its number of dimensions; a *slice* is a two-dimensional section of the tensor, defined by fixing all but two indices; and a *fiber* is a one-dimensional section obtained by fixing every index but one [17].

[2] We note that the permutations π_i could be restricted to only permute the non-null elements in the member set A_{i*}.

As an *inductive* embedding function, $\overline{\overline{f}}$ should be applicable to any SoS input $\mathbf{A}$ without imposing any constraints on its dimensions or values, as opposed to *transductive* embedding approaches.

The next proposition shows that inductive SoS embeddings cannot be simply represented as inductive set-embeddings (such as those studied in [6, 15, 25, 29, 40]) by some clever transformation of the input. We defer its proof to Appendix A.

Proposition 1. *There exists an inductive SoS embedding $\overline{\overline{f}}$ that cannot be represented as a (plain) set-embedding. Formally, for any permutation-invariant set-embedding $\overline{\overline{g}}$, and any encoding scheme σ (independent of $\overline{\overline{f}}$) that transforms an SoS $\mathbf{A}$ into a plain set $A = \sigma(\mathbf{A})$, there exists an SoS $\mathbf{A}$ such that $\overline{\overline{f}}(\mathbf{A}) \neq \overline{\overline{g}}(\sigma(\mathbf{A}))$.*

While set-embeddings are not expressive enough to represent set-of-sets, we propose an alternative representation motivated by the work of [25] and the concept of *Janossy densities* in the theory of point processes [9]. We can characterize any scalar- or vector-valued SoS permutation-invariant function as the average of another (possibly permutation-sensitive) SoS function over all possible member-set-level and element-level permutations:

Theorem 1. *Given a function $\vec{f}$ (which could be sensitive to input permutations) that maps an $n \times m \times d$ SoS tensor $\mathbf{A}$ to real- or vector-values, consider the function*

$$\overline{\overline{f}}(\mathbf{A}) = \frac{1}{n! \cdot (m!)^n} \sum_{\phi \in \Pi_n} \left[\sum_{\pi_1 \in \Pi_m} \cdots \sum_{\pi_n \in \Pi_m} \vec{f}(\mathbf{A}_{\phi,\pi_\phi}) \right], \quad (2)$$

where $\mathbf{A}_{\phi,\pi_\phi}$ denotes the tensor with (i, j)-th fiber $A_{i,j}$ equal to $A_{\phi(i),\pi_{\phi(i)}(j)}$. Then, $\overline{\overline{f}}$ is invariant to SoS permutations.

Furthermore, if $\vec{f}$ is modeled by a universal-approximator neural network (as defined in Hornik et al. [14]), then $\overline{\overline{f}}$ can approximate any inductive SoS embedding arbitrarily well.

The proof is deferred to Appendix A. Theorem 1 provides critical insight into how an inductive SoS embedding could be modeled. In particular, while it is intractable to directly model $\overline{\overline{f}}$, we could instead focus on tractable approaches for learning $\vec{f}$. Since $\vec{f}$ does not need to obey any SoS permutation-invariance constraints, we are free to parameterize $\vec{f}$ with any permutation-sensitive model. Thus, as long as we have a sufficiently expressive model for $\vec{f}$, we could *in principle* learn any SoS representation $\overline{\overline{f}}$ mapping an SoS $\mathbf{A}$ to a target value y (*e.g.*, class label in classification tasks or real-valued response in regression tasks). Before discussing how to apply the characterization given by Equation (2) in practice, let us examine its flexibility through a few examples.

3.2.2 *Examples of SoS embeddings.*

We discuss several examples of inductive SoS embeddings arising from concrete applications. In Section 5, we will demonstrate that these functions could be effectively learned in practice by exploiting Equation (2).

Basic set/multiset operations. Possibly the simplest examples of SoS representation functions on sets-of-sets are those that involve basic set/multiset operations, such as set union and intersection. These allow one to compute various population statistics for an SoS, such as counting or summing-up the (unique) elements in the union or intersection across all the member sets in an SoS.

Adamic/Adar index. In social network analysis, the *Adamic/Adar index* [1] is a simple and popular measure of the similarity between any two nodes in a network, which could be used to predict unseen links between nodes. For a node v in the network, denote its set of neighbors by N_v. The Adamic/Adar index between any two nodes u and v is defined as

$$g(u,v) = \sum_{x \in N_u \cap N_v} \frac{1}{\log |N_x|}, \tag{3}$$

where $|N_x|$ gives the *degree* of node x. Compared to other similarity measures such as the Jaccard coefficient, the Adamic/Adar index down-weights the importance of shared neighbors with very large neighborhoods. To see how Equation (3) could be cast in the form of Equation (2), let $\mathbf{A}^{(u,v)}$ be an SoS consisting of two sets A_u and A_v, corresponding to the neighborhoods of nodes u and v, respectively. Specifically, let $A_u = \{(x, |N_x|) : x \in N_u\}$ (and similarly for A_v) contain both the identifier and the degree of each neighboring node. Then, it is clear that $g(u,v)$ can be expressed as an inductive SoS embedding $\overline{\overline{g}}(\mathbf{A}^{(u,v)})$.

Multi-instance learning. In multi-instance learning [10, 22], the training data is a collection of labeled *bags*, each containing multiple instances. Thus, the learner seeks to learn a function f mapping a set to a class label. In many applications, such as predicting hyperlinks between subgraphs and anomaly detection with point-cloud scenes (see Section 5 for more details on both tasks), each instance within a bag is also a set, and the function to be learned is an inductive SoS embedding that should be invariant to permutations both within and across the instances in a bag.

4 LEARNING INDUCTIVE SOS EMBEDDINGS

In this section, we explore approaches to learning inductive SoS embeddings $\overline{\overline{f}}$, as described in Definition 1, that maps an input set-of-sets to a class label (in classification tasks) or real-valued response (in regression tasks). In particular, we shall exploit the characterization provided by Equation (2): rather than directly modeling $\overline{\overline{f}}$, we seek to learn the function $\vec{f}$, which gives us the freedom to apply any flexible family of models without being subject to the constraints of SoS permutation-invariance. In view of their expressiveness and flexibility as universal function approximators, we choose to model $\vec{f}$ using deep neural networks. In Section 4.1, we introduce our proposed neural network architecture; and in Section 4.2, we examine stochastic optimization methods for tractably learning $\vec{f}$, as well as Monte Carlo sampling procedures for transforming the learned $\vec{f}$ into an inductive SoS embedding $\overline{\overline{f}}$.

4.1 The HATS Architecture

We propose a hierarchical, sequence-attention architecture for learning inductive SoS embeddings. used in document classification [38] and adds two permutation layers, which are key ingredients in the tractable optimization and inference of HATS.

Background. Recurrent neural networks (RNNs) have been shown to be very well-suited for modeling functions over variable-length sequences. In particular, the use of parameter-sharing allows RNNs to simultaneously achieve flexibility and expressiveness in capturing complex interactions over variable-length sequences with only a fixed number of parameters. In fact, Siegelmann and Sontag [33] showed that (with exact computations) RNNs are *universal* functions in that any function computable by a Turing machine can be computed by an RNN of finite size. To alleviate the problem of vanishing or exploding gradients associated with capturing long-range dependencies, gated RNNs such as *long short-term memory networks* (LSTMs) and *gated recurrent units* (GRUs) have been proposed, and both have achieved great success in practical applications. We shall focus on LSTM-based architectures in this work.

We begin the description of HATS with a vanilla LSTM model for modeling SoS functions, and then propose more sophisticated designs that are tailored to the hierarchical nature of sets-of-sets.

4.1.1 *LSTM model for sets-of-sets.*
The SoS input $\mathbf{A}$ is given by a collection of sequences arranged in arbitrary order. We can model the function $\vec{f}$ in Equation (2) using an LSTM, which takes as input a single sequence obtained by concatenating all "rows" $\{\mathbf{A}_{i*}\}_{i=1}^{n}$ (we collapse consecutive null symbols into a single "#"). The last long-term memory state (or output state) of the LSTM is then fed into a multi-layer perceptron to obtain the final embedding.

4.1.2 *H-LSTM: Hierarchical LSTM model for sets-of-sets.*
By simply concatenating the constituent sequences within each set, the vanilla LSTM model discussed previously does not take into account the hierarchical nature of the sets-of-sets problem. In this section, we propose to use a two-level hierarchical LSTM (H-LSTM) model to capture the structure of sets-of-sets. Similar hierarchical LSTM architectures have been studied in the natural language processing literature (*e.g.*, document classification [38]).

Given an input SoS tensor $\mathbf{A}$ consisting of n sets (viewed as sequences) $\mathbf{A}_{i,*}$, $i = 1, \ldots, n$, with maximum cardinality m, the first layer of the H-LSTM model applies a bidirectional LSTM to each sequence $\mathbf{A}_{i,*}$ Specifically, let

$$\overrightarrow{\boldsymbol{h}_{ij}} = \overrightarrow{\mathrm{LSTM}_1}(A_{ij}), \quad \overleftarrow{\boldsymbol{h}_{ij}} = \overleftarrow{\mathrm{LSTM}_1}(A_{ij}), \quad j = 1, \ldots, m \tag{4}$$

denote the forward and backward hidden states for the j-th element in $\mathbf{A}_{i,*}$ obtained from the forward and backward LSTMs, respectively. We obtain an *annotation* for A_{ij} by concatenating the forward and backward hidden states: $\boldsymbol{h}_{ij} = \left[\overrightarrow{\boldsymbol{h}_{ij}}, \overleftarrow{\boldsymbol{h}_{ij}}\right]$, which summarizes the information pertaining to A_{ij} in the set $\mathbf{A}_{i,*}$. The last hidden state of the trained LSTM then provides an *embedding* of the whole set $\mathbf{A}_{i,*}$, which we denote as $\boldsymbol{h}_i$.

Existing approaches to modeling permutation-invariant functions use various forms of pooling operations (*e.g.*, max-pooling [27, 40] or Janossy pooling [25]) to aggregate the embeddings obtained for each element in a set. While these simple approaches are guaranteed to be invariant to permutations in the input, they do not allow flexibility to model complex interactions among the elements. Instead, we propose to concatenate the embeddings $\boldsymbol{h}_i$ obtained for each set $\mathbf{A}_{i,*}$, and then apply another bidirectional LSTM to model the dependencies among the set embeddings:

$$\overrightarrow{\boldsymbol{y}_i} = \overrightarrow{\mathrm{LSTM}_2}(\boldsymbol{h}_i), \; \overleftarrow{\boldsymbol{y}_i} = \overleftarrow{\mathrm{LSTM}_2}(\boldsymbol{h}_i), \; \boldsymbol{y}_i = \left[\overrightarrow{\boldsymbol{y}_i}, \overleftarrow{\boldsymbol{y}_i}\right], \; i = 1, \ldots, n. \tag{5}$$

The last hidden state of this upper-layer LSTM then provides an overall embedding $\boldsymbol{y}$ of the SoS $\mathbf{A}$ that takes into account its hierarchical structure. Finally, the target output (*e.g.*, class label in classification tasks) can be modeled with a fully connected layer using a softmax function.

4.1.3 HATS architecture. Capturing long-range dependencies is especially important in modeling functions over sets. Unlike language models, where adjacent words in a sentence often provide more information than words that are farther apart, the elements in a set are typically arranged in random order within a sequence,[3] and elements that appear in the early parts of the sequence contain information that is equally relevant to the final output embedding as those that are near the end.

Thus, when using a sequence model, such as an LSTM, to model a function over sets, it is essential to ensure that the model is able to capture both long-range and short-term dependencies. The same argument also applies to the set-level: since there is typically no canonical ordering for sets within an SoS, the top-level LSTM used in the H-LSTM model of the previous section should also be able to preserve long-range information in its final output embedding $\boldsymbol{y}$.

While LSTMs hypothetically should be able to capture long-range dependencies in sequences, in practice their performance are often less than ideal. Intuitively, requiring the last hidden state of an LSTM to encode information from a long input sequence into a single fixed-length vector seems too much to ask for. Such inability to capture long-range dependencies has aroused much concern in the natural language processing and machine translation communities, and many clever tricks (such as reversing the order of the input sequence) have been devised to improve their practical performance. However, when modeling functions over sets, these tricks are typically ineffective as the elements are arranged in random order in the input sequence.

Rather than attempting to encode a whole input sequence into a single fixed-bit vector (*i.e.*, the last hidden state), attention mechanisms [2] adaptively compute a weighted combination of all the hidden states during the decoding phase. By learning the weights in the attention mechanism, the decoder could then decide on which parts of the input sequence to focus on. This relieves the burden of having to preserve all information in the input sequence from the encoder, and allows the RNN to capture long-range dependencies.

In our context, different elements of a set may possess varying degrees of importance to the task at hand. For instance, when predicting the unique number of elements in a set (see Section 5.1 for more details), elements that occur very frequently may be regarded as less important than rare elements. Similarly, inside an SoS, smaller sets may contain less information than larger sets (or vice versa). To capture long-range dependencies on both element-level and set-level, we propose to adopt a *hierarchical attention* mechanism in a hierarchical bidirectional LSTM.

Element-level attention. Given an input SoS $\mathbf{A}$ comprising the sets $\mathbf{A}_{i,*}$, $i = 1, \dots, n$, let $\boldsymbol{h}_{ij}$ denote the annotation for the element A_{ij} in set $\mathbf{A}_{i,*}$ obtained by concatenating the forward and backward hidden states of Equation (4). We first pass $\boldsymbol{h}_{ij}$ through a feedforward layer with weights $\boldsymbol{W}_1$ and bias term b_1 to obtain a hidden representation of $\boldsymbol{h}_{ij}$:

$$\boldsymbol{u}_{ij} = \tanh(\boldsymbol{W}_1 \boldsymbol{h}_{ij} + b_1),$$

[3]With the exception of domain-specific scenarios where a *canonical* ordering could be imposed on the elements in the set; this is typically unavailable in general settings.

then compute the (normalized) similarity between $\boldsymbol{u}_{ij}$ and an element-level context vector $\boldsymbol{c}_1$ via

$$\alpha_{ij} = \frac{\exp(\boldsymbol{u}_{ij}^\mathsf{T} \boldsymbol{c}_1)}{\sum_{j'} \exp(\boldsymbol{u}_{ij'}^\mathsf{T} \boldsymbol{c}_1)},$$

which we use as importance weights to obtain the final embedding of the set $\mathbf{A}_{i,*}$:

$$\boldsymbol{h}_i = \sum_j \alpha_{ij} \boldsymbol{h}_{ij}. \tag{6}$$

Member-set-level attention. We feed the embeddings $\boldsymbol{h}_1, \dots, \boldsymbol{h}_n$ obtained from Equation (6) into the upper-level bidirectional LSTM and obtain the set annotations $\boldsymbol{y}_1, \dots, \boldsymbol{y}_n$ via Equation (5). Following a similar manner as in element-wise attention, we compute

$$\boldsymbol{v}_i = \tanh(\boldsymbol{W}_2 \boldsymbol{y}_i + b_2),$$
$$\beta_i = \frac{\exp(\boldsymbol{v}_i^\mathsf{T} \boldsymbol{c}_2)}{\sum_{i'} \exp(\boldsymbol{v}_{i'}^\mathsf{T} \boldsymbol{c}_2)},$$
$$\boldsymbol{y} = \sum_i \beta_i \boldsymbol{y}_i.$$

where $\boldsymbol{W}_2$ and b_2 are the weights and bias of another feedforward layer, $\boldsymbol{c}$ is a set-level context vector, β_i are importance weights, and $\boldsymbol{y}$ is the final embedding for SoS $\mathbf{A}$.

The overall *hierarchical attention* (HATS) model is illustrated in Figure 1. The LSTM with element-level attention encodes each set into a permutation-invariant embedding; the LSTM with member-set-level attention then computes a final permutation-invariant embedding for the SoS. The main difference from existing architectures [38] lies in the two permutation layers. The lower permutation layer performs an intra-set permutation for each set, while the upper layer performs an inter-set permutation. These two layers combine to ensure that the learned model is SoS permutation-invariant.

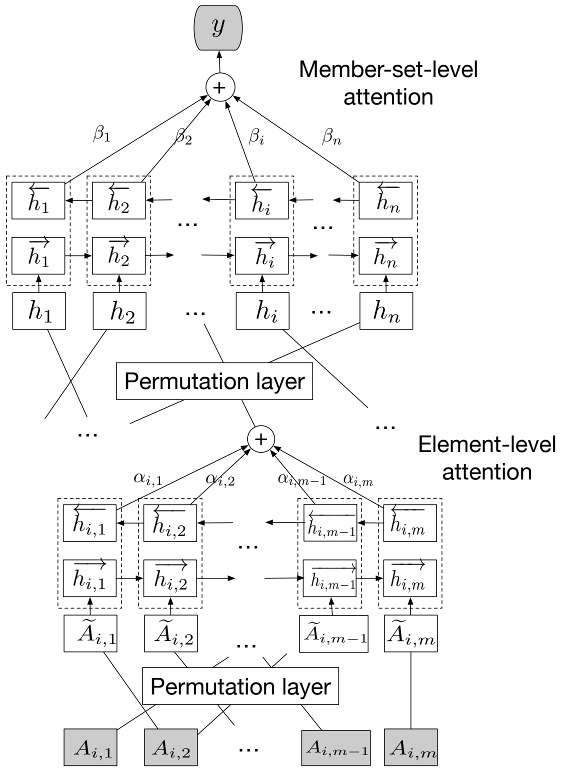
4.2 Stochastic Optimization for HATS

The obvious caveat in applying Equation (2) to learning $\overline{\overline{f}}$ in practice is that the $(n + 1)$ summations involved would be computationally prohibitive in all but the simplest scenarios, especially since each summation is over $n!$ (for ϕ) or $m!$ (for π_i, $i = 1, \dots, n$) possible permutations and thus, is already intractable for moderate n and m. To learn HATS, we adapt the stochastic optimization procedure (π-SGD) of [25] to set-of-sets inputs.

Consider N labeled SoS training examples $\{(\mathbf{A}^{(s)}, y_s)\}_{s=1}^N$, where $\mathbf{A}^{(s)}$ is a set-of-sets and y_s is its label (class label in classification tasks or real-valued response for regression tasks). Let $\hat{y}$ be the predicted label for the SoS input $\mathbf{A}$. Consider a loss function $L(y, \hat{y})$ such as squared-loss or cross-entropy loss. In general, L only needs to be convex in $\hat{y}$, but it does *not* need to be convex with respect to the neural network parameters $\boldsymbol{\theta} = (\boldsymbol{W}_1, b_1, \boldsymbol{W}_2, b_2, \boldsymbol{\theta}_1, \boldsymbol{\theta}_2)$, where $\boldsymbol{\theta}_1$ and $\boldsymbol{\theta}_2$ are the parameters of the two bidirectional-LSTMs.

We wish to learn an SoS function $\overline{\overline{f}}(\,\cdot\,; \boldsymbol{\theta})$ with parameters $\boldsymbol{\theta}$ that minimizes the empirical risk on the training data:

$$\boldsymbol{\theta}^* = \arg\min_{\boldsymbol{\theta}} \sum_{s=1}^N L\left(y_s, \overline{\overline{f}}(\mathbf{A}^{(s)}; \boldsymbol{\theta})\right). \tag{7}$$



y
Member-set-level attention
β_1 β_2 β_i β_n
$\overleftarrow{h}_1$ $\overleftarrow{h}_2$ $\overleftarrow{h}_i$ $\overleftarrow{h}_n$
$\overrightarrow{h}_1$ $\overrightarrow{h}_2$ $\overrightarrow{h}_i$ $\overrightarrow{h}_n$
h_1 h_2 h_i h_n
Permutation layer
Element-level attention
$\alpha_{i,1}$ $\alpha_{i,2}$ $\alpha_{i,m-1}$ $\alpha_{i,m}$
$\widetilde{A}_{i,1}$ $\widetilde{A}_{i,2}$ $\widetilde{A}_{i,m-1}$ $\widetilde{A}_{i,m}$
Permutation layer
$A_{i,1}$ $A_{i,2}$ $A_{i,m-1}$ $A_{i,m}$

Figure 1: HATS architecture for SoS inputs.

Naturally, $\overline{\overline{f}}$ should satisfy Definition 1, since the input set-of-sets is invariant under SoS permutations. To avoid evaluating $\overline{\overline{f}}(\mathbf{A};\boldsymbol{\theta})$ (which involves an intractable summation over all permutations in Equation (2)), we will stochastically minimize an upper bound to the objective in Equation (7).

Our optimization procedure is described in Algorithm 1. Rather than summing over all permutations, we sample orderings uniformly at random from the space of permutations: $\widetilde{\phi} \sim \text{Uniform}(\Pi_n)$, $\widetilde{\pi}_1, \ldots, \widetilde{\pi}_n \sim \text{Uniform}(\Pi_m)$, and compute

$$\hat{\overline{\overline{f}}}(\mathbf{A}) = \vec{f}(\mathbf{A}_{\widetilde{\phi}, \widetilde{\pi}(\widetilde{\phi})}) \,. \quad (8)$$

It is easy to see that Equation (8) provides an unbiased estimator:

$$\mathbb{E}_{\widetilde{\phi}, \widetilde{\pi}_1, \ldots, \widetilde{\pi}_n}[\hat{\overline{\overline{f}}}(\mathbf{A};\boldsymbol{\theta})] = \overline{\overline{f}}(\mathbf{A};\boldsymbol{\theta}) \,.$$

For the s-th training example, using the sampled permutations, Algorithm 1 can be shown to optimize

$$\mathbb{E}_{\widetilde{\phi}, \widetilde{\pi}_1, \ldots, \widetilde{\pi}_n}[L(y_s, \hat{\overline{\overline{f}}}(\mathbf{A}^{(s)};\boldsymbol{\theta}))] \,,$$

Since $L(y_s, \cdot)$ is a convex function, by Jensen's inequality,

$$\mathbb{E}_{\widetilde{\phi}, \widetilde{\pi}_1, \ldots, \widetilde{\pi}_n}[L(y_s, \hat{\overline{\overline{f}}}(\mathbf{A}^{(s)};\boldsymbol{\theta}))] \geq L(y_s, \overline{\overline{f}}(\mathbf{A}^{(s)};\boldsymbol{\theta})) \,.$$

Hence, Algorithm 1 optimizes a proper surrogate to the original objective in Equation (7). In practice, one could also sample multiple permutations and average over them in Equation (8) to reduce the variance of the estimator.

The computational cost of optimizing HATS lies in backpropagating the gradients through the neural network architecture (*cf.* Figure 1) of the HATS model $\vec{f}$. Thus, the overall time complexity of Algorithm 1 is equal to $O(mndTB)$. At inference time, we perform a Monte Carlo estimate of Equation (2) by sampling a few permutations and performing a forward pass over the HATS neural network. Our experiments show that in practice five to twenty Monte Carlo samples are sufficient for estimating Equation (2).

Algorithm 1: Stochastic optimization for learning HATS.

```
Input: Labeled SoS training examples $\{(\mathbf{A}^{(s)}, y_s)\}_{s=1}^N$;
Input: HATS model $\vec{f}(\mathbf{A};\boldsymbol{\theta})$ with unknown parameters $\boldsymbol{\theta}$;
Input: Loss function $L(y, \hat{y})$;
Input: Number of optimization epochs $T$;
Input: Mini-batch size $B$; learning-rate schedule $\{\eta_t\}_{t=1}^T$;
Output: Learned parameters $\boldsymbol{\theta}$ for the model $\vec{f}(\mathbf{A};\boldsymbol{\theta})$.
1  Initialize parameters $\boldsymbol{\theta}^{(0)}$ ;
2  for $t = 1, \ldots, T$ do
3      $\boldsymbol{g}_t \leftarrow \mathbf{0}$ ;
4      for $s$ in mini-batch-indices do
5          $\widetilde{\mathbf{A}}^{(s)} \leftarrow$ Permute the rows of $\mathbf{A}^{(s)}$ ;
6          for $i = 1, \ldots, |\widetilde{\mathbf{A}}^{(s)}|$ do
7              Permute the entries of $\widetilde{\mathbf{A}}_{i*}^{(s)}$ ;
8          $\boldsymbol{g}_t \leftarrow \boldsymbol{g}_t + \frac{1}{B}\nabla_{\boldsymbol{\theta}} L(y_s, \vec{f}(\widetilde{\mathbf{A}}^{(s)};\boldsymbol{\theta}))$ ;
9      $\boldsymbol{\theta}^{(t)} \leftarrow \boldsymbol{\theta}^{(t-1)} - \eta_t \boldsymbol{g}_t$ ;
10 return $\boldsymbol{\theta}^{(T)}$
```

5 EXPERIMENTS

We demonstrate the utility of HATS and the stochastic optimization procedure by conducting experiments on a variety of SoS tasks spanning multiple applications, ranging from arithmetic tasks and computing similarity measures between sets, to predicting hyperlinks across subgraphs in large networks, to detecting anomalous point-clouds in computer vision.

Models. We evaluate the performance of our proposed models and compare with several existing approaches in the literature (*cf.* Section 2). We summarize all the models below:

DeepSet [40] : A feedward neural network model with sum-pooling to achieve permutation-invariance.
MI-CNN [15] : Convolutional network with gated attention mechanisms for permutation-invariant multi-instance learning.
J-LSTM [25]: The vanilla LSTM model of Section 4.1.1; equivalent to applying the Janossy pooling method [25] using LSTMs.
H-LSTM : The hierarchical LSTM of Section 4.1.2.
HATS : The hierarchical attention network of Section 4.1.3.

Since DeepSet [40], MI-CNN [15], and J-LSTM [25] were originally designed for set (rather than SoS) inputs, we flatten each SoS by concatenating its member sets into a single sequence as a preprocessing step. The J-LSTM, H-LSTM, and HATS models are trained using the framework described in Algorithm 1. The runtime complexities of all methods are proportional to the size of the input data:

$O(mnd)$. For DeepSet [40], MI-CNN [15], and J-LSTM [25] we adopt the authors' implementations.[4] Details on the models and training procedures are provided in Appendix B; code for reproducing all experiments can be found at `github.com/PurdueMINDS/HATS`.

5.1 Simple Arithmetic Tasks

Similar to [25, 40], we begin by considering simple arithmetic tasks that involve predicting summary statistics for sets-of-sets containing integers. In our experiments, each SoS contains $n = 4$ member multisets, created by drawing m integers from $\{0, 1, \ldots, 9\}$ with replacement. Given $N = 10{,}000$ SoS training examples, we predict:

$\cap$ *Binary*: Whether the intersection of all member sets is empty.
$\cap$ *Sum*: Sum of all elements in the intersection of all member sets.
$\cup$ *Sum*: Sum of all elements in the union of all member sets.
Unique count: Number of unique elements across all member sets.
Unique sum: Sum of all unique elements across all member sets.

Note that the first three tasks ($\cap$ *Binary*, $\cap$ *Sum*, and $\cup$ *Sum*) are *interactive* in that they require learning interactions across the member sets of an SoS, while the last two tasks (*Unique count* and *Unique sum*) are *aggregative* in that their results would remain unchanged if one simply concatenated all the member sets into a single set and computed the unique count/sum of its elements.

For each model, we use a validation set containing another 10,000 examples and evaluate its predictions on a held-out test set with 10,000 examples. For each task, we also experiment with two different member-set sizes m. For each method, we conduct five random trials and report the mean and standard deviations of their test-set prediction accuracies. The results are shown in Tables 1 and 2; for each task, we indicate the highest accuracy values (within two standard errors) in boldface.

We observe that the sequence models (J-LSTM, H-LSTM, HATS) significantly outperform MI-CNN and DeepSet on most tasks. This shows that RNNs (learned with stochastic optimization) are more suitable for modeling variable length inputs than CNNs or sum-pooling (DeepSet). We also note that H-LSTM substantially outperforms J-LSTM, which shows that modeling the hierarchical structure of set-of-sets can better capture and decouple the inter-set and intra-set dependencies. Furthermore, we observe that HATS performs on par with or superior to H-LSTM, demonstrating the effectiveness of the element-level and set-level attention mechanisms in capturing higher-range dependencies within and across member sets.

SoS improvement over aggregative tasks. The *aggregative* tasks (such as *Unique count* and *Unique sum*) are essentially set tasks disguised as SoS tasks, since concatenating all the member sets into a single set suffices for these tasks. Intuitively, DeepSet, MI-CNN, and J-LSTM should perform well on these tasks since the hierarchical structure of SoS's do not play a role in how the true label was generated. However, from Table 2 we still observe that by modeling what is essentially a set function as an SoS function, H-LSTM and HATS are still able to produce significant gains over the other approaches. We believe that this is due to the fact that sequence models like LSTMs still have trouble capturing long-range dependencies—by segmenting a single long sequence into a collection of short sequences (*i.e.*, modeling a set as an SoS), one could improve the models' capability of capturing dependencies among elements.

5.2 Computing the Adamic/Adar Index

In Section 3.2.2, we showed that the Adamic/Adar (A/A) index [1] between two nodes in a network could be cast as an SoS function. We perform experiments on the `Cora` [32] dataset,[5] in which we evaluate J-LSTM, H-LSTM, and HATS approaches for predicting the Adamic/Adar index between pairs of nodes using their neighbor sets. More specifically, as described in Section 3.2.2, each input SoS contains two neighbor sets; an element in each set is a tuple containing the unique identifier of neighboring node and its degree. Since computing the A/A index requires both node identifier and degree information, it is not straightforward to transform the SoS inputs into a single set. Thus, we only evaluate the A/A task over methods that can operate with native SoS inputs. An added difficulty is the variable number of neighbors of nodes.

We use $N = 1{,}000$ randomly sampled node-pairs for training, 1,000 for validation, and another 1,000 for held-out testing. We compute the predicted A/A values for held-out test examples and measure the mean-absolute error (MAE) and mean-squared error (MSE) between the model predictions and the true A/A index. The results are shown in Table 3.[6] We observe that HATS attains the lowest errors, followed by H-LSTM and then J-LSTM. We note that computing the A/A index implicitly requires calculating the intersection of two sets, which as our previous task has shown (*cf.* Table 1), both HATS and H-LSTM perform well.

5.3 Subgraph Hyperlink Prediction

Modeling higher-order structures within a network have recently attracted attention in relational learning and graph mining (*e.g.*, [4, 21]). Here, our task is to learn SoS embeddings that can help predict the existence of *hyperlinks* between subgraphs of a large network. Specifically, a hyperlink exists between two subgraphs if there is at least one link connecting two nodes from different m-node induced subgraphs in a larger graph.

We perform experiments on three widely used network datasets: the citation network `Cora` [32], the Wikipedia voting network `Wiki-vote` in [20] and the protein-protein interaction network `PPI` in [12]. Table 7 in Appendix 7 provides a summary of the network statistics. For each network, we first obtain $d = 256$-dimensional feature representations for each node using unsupervised GraphSAGE [12]. Given an SoS example consisting of the features of every node in each subgraph, the task is to predict a binary label indicating whether a hyperlink exists between the two subgraphs in that SoS. [7] For each dataset, we also vary the subgraph size m.

Table 4 shows the hyperlink prediction accuracies for each network dataset. We observe that HATS and H-LSTM outperform the other approaches in all tasks.

[4] https://github.com/manzilzaheer/DeepSets
https://github.com/AMLab-Amsterdam/AttentionDeepMIL
https://github.com/PurdueMINDS/JanossyPooling

[5] Table 7 in Appendix C summarizes the dataset statistics.
[6] For evaluation, we only consider node-pairs whose neighborhoods overlap, since the A/A index is trivially zero for node-pairs with disjoint neighborhoods.
[7] Since the `PPI` network is much denser, we instead perform the multi-class classification task of predicting the number of hyperlinks between subgraphs.

Table 1: Prediction accuracies for *interactive* arithmetic tasks for different member-set size m.

	$m = 5$			$m = 10$		
Methods	$\cap$ *Binary*	$\cap$ *Sum*	$\cup$ *Sum*	$\cap$ *Binary*	$\cap$ *Sum*	$\cup$ *Sum*
DeepSet [40]	**0.742 (0.005)**	**0.765 (0.003)**	0.078 (0.003)	0.900 (0.003)	0.069 (0.003)	0.873 (0.003)
MI-CNN [15]	**0.741 (0.007)**	0.739 (0.006)	0.425 (0.111)	**0.904 (0.001)**	0.071 (0.003)	0.873 (0.071)
J-LSTM [25]	0.729 (0.005)	**0.762 (0.003)**	0.956 (0.001)	0.271 (0.001)	0.599 (0.006)	0.870 (0.003)
H-LSTM	0.736 (0.003)	**0.763 (0.002)**	0.963 (0.061)	0.903 (0.002)	0.967 (0.009)	0.893 (0.012)
HATS	**0.740 (0.007)**	**0.765 (0.002)**	**0.996 (0.005)**	**0.904 (0.003)**	**0.998 (0.001)**	**0.925 (0.012)**

5.4 Point-Cloud SoS Classification

Point-clouds (*i.e.*, sets of low-dimensional vectors in Euclidean space) arise in many computer vision applications such as autonomous driving using LIDAR data [27]. We perform experiments on the `ModelNet40` [36] point-cloud database which contains more than 12,311 point-clouds, each labeled as one of 40 classes (such as *desk*, *chair*, or *plane*; see Figure 3 for some example visualizations) .

In our experiments, each SoS example, representing a point-cloud scene, contains $m = 10$ point-clouds, and each point-cloud comprises 2,000 points. We construct SoS examples in two different ways to perform two prediction tasks:

Anomaly detection: Among the 10 point-clouds in each SoS, at least 9 of them have the same class label, but there is a 50% chance that the remaining one has a different label (*i.e.*, is an anomaly). Given such an SoS, the task is to predict whether this SoS contains an anomalous point-cloud.

Unique-label counting: Each SoS consists of 10 randomly selected point-clouds from the database. Given such an SoS, the task is to predict the number of unique object types (labels) in the SoS.

For each task, we use $N = 2{,}000$ SoS examples for training, 1,000 for validation, and 2,000 held-out for testing. Table 5 shows the prediction accuracies for each method on both tasks. Once again, we observe that HATS performs best among all approaches.

To further gauge the relative performance of the sequence models, Figure 3 varies the size m of each point-cloud (*i.e.*, the number of points it contains), and plot the resulting accuracies (along with standard errors) for the anomaly detection task in Figure 3. We observe that HATS consistently outperforms H-LSTM and J-LSTM, thanks to its attention mechanism for capturing long-range dependencies even as m gets to 1,500 elements in the member sets.

For SoS's whose member sets are rather large (for instance, each point-cloud instance contains $m = 2{,}000$ points), one could further speed up the training procedure of our proposed models by retaining only the first k columns of the permuted SoS $\widetilde{\mathbf{A}}$ from m to a smaller number k after line 5 of Algorithm 1. This approach can be viewed as an example of imposing *k-ary dependency restrictions* [25] to promote computational efficiency. For the point-cloud anomaly detection task, Figure 3 investigates how such k-ary restrictions affect prediction performance. We observe that even with small values of k, the loss in prediction accuracy remains tolerable, even as k decreases from the original $m = 2{,}000$ points (Table 5) to 50 points.

	$m = 20$		$m = 40$	
Methods	*Unique count*	*Unique sum*	*Unique count*	*Unique sum*
DeepSet [40]	0.432 (0.009)	0.080 (0.002)	0.858 (0.002)	0.872(0.002)
MI-CNN [15]	0.071 (0.003)	0.873 (0.071)	0.860 (0.003)	0.876 (0.002)
J-LSTM [25]	0.942 (0.003)	0.955 (0.001)	0.858 (0.001)	0.872 (0.004)
H-LSTM	0.988 (0.007)	0.991 (0.002)	0.892 (0.007)	0.948 (0.069)
HATS	**0.996 (0.006)**	**0.996 (0.007)**	**0.938 (0.03)**	**0.998 (0.002)**

Table 2: Accuracies for *aggregative* tasks with different member-set size m.

Models	MAE	MSE
LSTM	0.109 (0.001)	0.129 (0.005)
H-LSTM	0.107 (0.004)	0.120 (0.007)
HATS	**0.103 (0.002)**	**0.108 (0.009)**

Table 3: Predicting Adamic/Adar-index on `Cora`.

6 CONCLUSION

In this work, we formalized the problem of learning inductive embeddings over set-of-sets (SoS) inputs, and showed that embeddings for sets [25, 40] are not expressive enough to be directly used for modeling sets-of-sets. We then proposed a framework for learning permutation-invariant inductive SoS embeddings with neural networks, and introduced HATS, a hierarchical sequence-attention architecture with permutation layers, that is designed to better capture intra-set and inter-set interactions in sets-of-sets while maintaining SoS permutation-invariance. We developed stochastic optimization and inference procedures for HATS, and demonstrated its superior performance over a wide range of application tasks involving SoS inputs.

ACKNOWLEDGMENTS

This research is supported by the NSF, AFRL, ARO, and DOD/SERC under contract numbers IIS-1618690, IIS-1546488, CCF-0939370, FA8650-18-2-7879, W911NF-09-2-0053, HQ0034-13-D-0004 RT #206, the Purdue Integrative Data Science Initiative, and the Purdue Research Foundation.

REFERENCES

[1] Lada A. Adamic and Eytan Adar. 2001. Friends and Neighbors on the Web. *Social Networks*.

[2] Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. 2014. Neural Machine Translation by Jointly Learning to Align and Translate. *arXiv:1409.0473*.

[3] 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. 2018. Relational inductive biases, deep learning, and graph networks. *arXiv:1806.01261*.

[4] Austin R Benson, Rediet Abebe, Michael T Schaub, Ali Jadbabaie, and Jon Kleinberg. 2018. Simplicial closure and higher-order link prediction. *PNAS*.

[5] Wayne D. Blizard. 1988. Multiset theory. *Notre Dame J. Formal Logic* 30.

[6] Benjamin Bloem-Reddy and Yee Whye Teh. 2019. Probabilistic symmetry and invariant neural networks. *arXiv:1901.06082*.

Table 4: Subgraph hyperlink prediction accuracies for different subgraph size m.

	$m = 4$			$m = 10$		
Methods	`Wiki-vote`	`Cora`	`PPI`	`Wiki-vote`	`Cora`	`PPI`
DeepSet [40]	0.657 (0.028)	0.676 (0.018)	0.731 (0.031)	0.638 (0.017)	0.661 (0.021)	0.431 (0.027)
MI-CNN [15]	0.526 (0.006)	0.938 (0.006)	0.782 (0.010)	0.473 (0.012)	0.940 (0.004)	0.764 (0.003)
J-LSTM [25]	0.518 (0.029)	0.936 (0.002)	0.465 (0.001)	0.517 (0.030)	0.938 (0.003)	0.410 (0.004)
H-LSTM	**0.741 (0.017)**	0.945 (0.005)	**0.910 (0.002)**	**0.743 (0.006)**	0.943 (0.004)	**0.801 (0.006)**
HATS	**0.740 (0.008)**	**0.950 (0.004)**	**0.913 (0.004)**	**0.742 (0.006)**	**0.952 (0.001)**	0.792 (0.006)

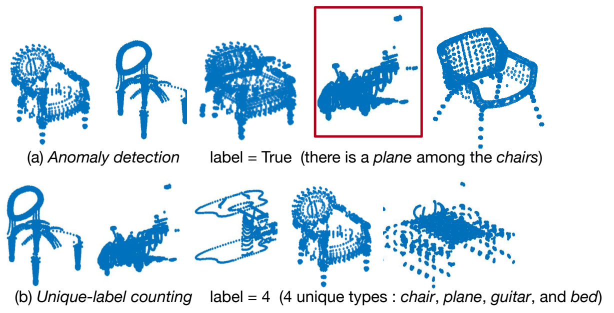

(a) Anomaly detection
label = True (there is a plane among the chairs)
(b) Unique-label counting
label = 4 (4 unique types : chair, plane, guitar, and bed)

Figure 2: Visualization of point-cloud tasks.

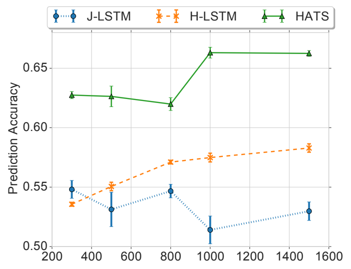


Figure 3: Anomaly detection accuracy for varying point-cloud size m.

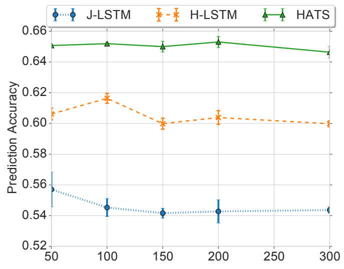


Figure 4: Anomaly detection accuracy for varying k-ary dependency.

Methods	*Anomaly detection*	*Unique-label counting*
DeepSet [40]	0.553 (0.021)	0.306 (0.039)
MI-CNN [15]	0.513 (0.003)	0.356 (0.011)
J-LSTM [25]	0.525 (0.018)	0.369 (0.009)
H-LSTM	0.602 (0.003)	0.357 (0.008)
HATS	**0.663 (0.016)**	**0.388 (0.011)**

Table 5: Point-cloud classification results.

[7] Michael M. Bronstein, Joan Bruna, Yann LeCun, Arthur Szlam, and Pierre Vandergheynst. 2017. Geometric Deep Learning: Going beyond Euclidean data. *IEEE Signal Process. Mag.* 34, 4, 18–42.

[8] Andrew Cotter, Maya Gupta, Heinrich Jiang, James Muller, Taman Narayan, Serena Wang, and Tao Zhu. 2018. Interpretable Set Functions. *arXiv:1806.00050.*

[9] D. J. Daley and D. Vere-Jones. 2008. *An Introduction to the Theory of Point Processes (Vol. II)* (second ed.). Springer.

[10] Thomas G. Dietterich, Richard H. Lathrop, and Tomás Lozano-Pérez. 1997. Solving the Multiple Instance Problem with Axis-parallel Rectangles. *Artif. Intell.* 89, 1-2 (1997), 31–71.

[11] Aditya Grover and Jure Leskovec. 2016. node2vec: Scalable feature learning for networks. In *KDD.*

[12] Will Hamilton, Zhitao Ying, and Jure Leskovec. 2017. Inductive representation learning on large graphs. In *NeurIPS.*

[13] Jason Hartford, Devon R Graham, Kevin Leyton-Brown, and Siamak Ravanbakhsh. 2018. Deep Models of Interactions Across Sets. In *ICML.*

[14] Kurt Hornik, Maxwell Stinchcombe, and Halbert White. 1989. Multilayer feedforward networks are universal approximators. *Neural networks* 2, 5, 359–366.

[15] Maximilian Ilse, Jakub M Tomczak, and Max Welling. 2018. Attention-based deep multiple instance learning. In *ICML.*

[16] Diederik P Kingma and Jimmy Ba. 2014. Adam: A method for stochastic optimization. *arXiv:1412.6980.*

[17] Tamara G. Kolda and Brett W. Bader. 2009. Tensor Decompositions and Applications. *SIAM Rev.* 51, 3 (2009), 455–500.

[18] Gil Lederman, Markus N Rabe, Edward A Lee, and Sanjit A Seshia. 2018. Learning Heuristics for Automated Reasoning through Reinforcement Learning.

[19] Juho Lee, Yoonho Lee, Jungtaek Kim, Adam R Kosiorek, Seungjin Choi, and Yee Whye Teh. 2018. Set Transformer. *arXiv:1810.00825* (2018).

[20] Jure Leskovec, Daniel Huttenlocher, and Jon Kleinberg. 2010. Signed networks in social media. In *CHI.*

[21] Dong Li, Zhiming Xu, Sheng Li, and Xin Sun. 2013. Link prediction in social networks based on hypergraph. In *WWW.*

[22] Oded Maron and Tomás Lozano-Pérez. 1997. A Framework for Multiple-instance Learning. In *NeurIPS.*

[23] Changping Meng, S Chandra Mouli, Bruno Ribeiro, and Jennifer Neville. 2018. Subgraph Pattern Neural Networks for High-Order Graph Evolution Prediction. In *AAAI.*

[24] John Moore and Jennifer Neville. 2017. Deep Collective Inference. In *AAAI.* 2364–2372.

[25] Ryan L. Murphy, Balasubramaniam Srinivasan, Vinayak Rao, and Bruno Ribeiro. 2019. Janossy pooling: Learning deep permutation-invariant functions for variable-size inputs. In *ICLR.*

[26] Mathias Niepert, Mohamed Ahmed, and Konstantin Kutzkov. 2016. Learning convolutional neural networks for graphs. In *ICML.*

[27] Charles R Qi, Hao Su, Kaichun Mo, and Leonidas J Guibas. 2017. Pointnet: Deep learning on point sets for 3d classification and segmentation. In *CVPR.*

[28] Charles Ruizhongtai Qi, Li Yi, Hao Su, and Leonidas J Guibas. 2017. Pointnet++: Deep hierarchical feature learning on point sets in a metric space. In *NeurIPS.*

[29] Siamak Ravanbakhsh, Jeff Schneider, and Barnabas Poczos. 2016. Deep Learning with Sets and Point Clouds. *arxiv:1611.04500* (2016).

[30] Adam Santoro, David Raposo, David G Barrett, Mateusz Malinowski, Razvan Pascanu, Peter Battaglia, and Timothy Lillicrap. 2017. A simple neural network module for relational reasoning. In *NeurIPS.*

[31] Daniel Selsam, Matthew Lamm, Benedikt Bunz, Percy Liang, Leonardo de Moura, and David L Dill. 2018. Learning a SAT Solver from Single-Bit Supervision. *arXiv:1802.03685.*

[32] Prithviraj Sen, Galileo Namata, Mustafa Bilgic, Lise Getoor, Brian Galligher, and Tina Eliassi-Rad. 2008. Collective classification in network data. *AI magazine.*

[33] Hava T. Siegelmann and Eduardo D. Sontag. 1991. Turing Computability With Neural Nets. *Appl. Math. Lett.* 4 (1991), 77–80.

[34] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention is All you Need. In *NeurIPS.*

[35] Oriol Vinyals, Samy Bengio, and Manjunath Kudlur. 2015. Order matters: Sequence to sequence for sets. In *ICLR.*

[36] Zhirong Wu, Shuran Song, Aditya Khosla, Fisher Yu, Linguang Zhang, Xiaoou Tang, and Jianxiong Xiao. 2015. 3D Shapenets: A deep representation for volumetric shapes. In *CVPR.*

[37] Keyulu Xu, Weihua Hu, Jure Leskovec, and Stefanie Jegelka. 2019. How Powerful are Graph Neural Networks?. In *ICLR.*

[38] Zichao Yang, Diyi Yang, Chris Dyer, Xiaodong He, Alexander J. Smola, and Eduard H. Hovy. 2016. Hierarchical Attention Networks for Document Classification. In *NAACL-HLT.*

[39] Seungil You, David Ding, Kevin Canini, Jan Pfeifer, and Maya Gupta. 2017. Deep lattice networks and partial monotonic functions. In *NeurIPS.*

[40] Manzil Zaheer, Satwik Kottur, Siamak Ravanbakhsh, Barnabás Póczos, Ruslan Salakhutdinov, and Alexander J. Smola. 2017. Deep Sets. In *NeurIPS.*

A PROOFS

Proof of Proposition 1. Consider an $n \times m \times d$ SoS tensor $\mathbf{A}$ with $n, m > 1$. Let $\overset{=}{f}$ be an SoS embedding function that computes the maximum member-set size: $\overset{=}{f}(\mathbf{A})$ equals the maximum number of non-null elements in $\mathbf{A}_{i,*}$ across all $i = 1, \ldots, n$. Given any encoding scheme σ (that does not utilize prior information in $\overset{=}{f}$), consider the (plain) set $A = \sigma(\mathbf{A})$ obtained via the encoding. If A does not contain null "#" symbols marking the end of each member set, clearly the cardinality of each member set is irrecoverably lost. If A retains null "#" symbols, because the output of any set-embedding $\overline{\overline{g}}(A)$ needs to be invariant with respect to permutations of the elements in A, $\overline{\overline{g}}(A)$ could not depend on the locations of "#" within A, and any information regarding the member-set cardinalities are once again lost. □

Proof of Theorem 1. It is straightforward to verify that the function $\overset{=}{f}$ defined in Equation (2) satisfies the requirement of Definition 1. Next, we show that if $\vec{f}$ is modeled by a universal-approximator neural network, then $\overset{=}{f}(\mathbf{A})$ can approximate any inductive SoS embedding of $\mathbf{A}$ arbitrarily well. We proceed via proof-by-contradiction.

Suppose that there exists an inductive SoS embedding $\overset{=}{f}'$ that is not universally approximated by $\overset{=}{f}$. Define a permutation-sensitive function $\vec{f}'$ satisfying

$$\vec{f}'(\mathbf{A}_{\phi,\pi_\phi}) = \overset{=}{f}'(\mathbf{A}) + e_{\phi,\pi_\phi}, \quad \forall \phi \in \Pi_n,\ \pi_1, \ldots, \pi_n \in \Pi_m,$$

where the "residuals" e_{ϕ,π_ϕ} are chosen such that $\sum_{\phi \in \Pi_n} \left[\sum_{\pi_1 \in \Pi_m} \cdots \sum_{\pi_n \in \Pi_m} e_{\phi,\pi_\phi} \right] = 0$. Then, $\vec{f}'$ cannot be universally approximated by $\vec{f}$, which contradicts the fact that $\vec{f}$ is an universal approximator of permutation-sensitive functions, thus concluding the proof. □

B IMPLEMENTATION DETAILS

All models are implemented using Python 3.6 with PyTorch 1.0. The LSTM cells used in J-LSTM [25], H-LSTM, and HATS all have 20-dimensional hidden states. The mini-batch size in Algorithm 1 is set to 32 for point-cloud classification tasks, and 128 for all other tasks. DeepSet [40] was originally proposed to handle (plain) set-inputs rather than SoS inputs. We use the `Adder` function in the authors' code to aggregate the embedding of all the member sets within an SoS into a single embedding.

For all the LSTM-based models, we use the Adam optimizer [16] with initial learning-rate 0.001. For each method/task, we use the validation set to select the best model as follows: during the training process, we retain the model that achieves the best validation metrics on the validation set and use it for testing. The validation metrics, as well as hyper-parameter values and loss functions are summarized in Table 6. Following [25, 40], we treat the simple arithmetic task as a regression task using L_1-loss. Since the true values of the arithmetic tasks are integers, we round the regression outputs to the nearest integers before computing the prediction accuracies.

Table 6: Implementation details for various tasks.

	Arithmetic tasks	Adamic/Adar index	Subgraph hyperlink prediction	Point-cloud classification
Task type	Regression	Regression	Classification	Classification
Loss function	L_1	L_1	Cross-entropy	Cross-entropy
Max. num. epochs	4000	4000	4000	2000
Validation metric	Accuracy	L_1-loss	Accuracy	Accuracy
Num. training examples	10,000	10,000	10,000	2,000
Num. validation examples	10,000	10,000	10,000	1,000
Num. test examples	10,000	10,000	10,000	2,000

C DATASET STATISTICS

Table 7 summarizes the statistics of the network datasets used in Section 5.3.

Table 7: Summary of network dataset statistics.

Dataset	$\lvert V\rvert$	$\lvert E\rvert$	#Classes
`Cora` [32]	2,708	5,429	7
`Wiki-vote` [20]	7,115	103,689	1
`PPI` [12]	3,890	76,584	50