Improving Graph Neural Network Representations of Logical Formulae with Subgraph Pooling

Maxwell Crouse‡, Ibrahim Abdelaziz†, Cristina Cornelio‡, Veronika Thost‡, Lingfei Wu‡, Kenneth Forbus§, Achille Fokouë‡

§Northwestern University, ‡IBM Research, †MIT-IBM Watson AI Lab

Abstract—Recent advances in the integration of deep learning with automated theorem proving have centered around the representation of logical formulae as inputs to deep learning systems. In particular, there has been a shift from character and token-level representations to graph-structured representations, in large part driven by the rapidly emerging body of research on geometric deep learning. Typically, structure-aware neural methods for embedding logical formulae have been variants of either Tree LSTMs or GNNs. While more effective than character and token-level approaches, such methods have often made representational trade-offs that limited their ability to effectively represent the global structure of their inputs. In this work, we introduce a novel approach for embedding logical formulae using DAG LSTMs that is designed to overcome the limitations of both Tree LSTMs and GNNs. The effectiveness of the proposed framework is demonstrated on the tasks of premise selection and proof step classification where it achieves the state-of-the-art performance on two standard datasets.

I. INTRODUCTION

While state-of-the-art classical theorem provers excel at finding complex proofs in restricted domains (e.g. TPTP [1]), they have historically had difficulty when reasoning in broad contexts [2], [3]. With the generation of large logical theories (collections of axioms) for reasoning [4]–[6], there has been interest in extending traditional theorem provers to handle the computational challenges inherent to reasoning at scale.

A natural way to address this challenge has been to design sophisticated mechanisms that allow theorem provers to determine which axioms or intermediate proof outputs merit exploration in the proof search process. These mechanisms thus pruned an otherwise unmanageably large proof search space down to a size that can be handled efficiently by classical theorem provers. The task of classifying axioms as being relevant to a given conjecture is referred to as premise selection, while the task of classifying intermediate proof steps as being a part of the final proof to a given conjecture is referred to as proof step classification.

Traditional approaches to solving these two tasks would generally rely on manually designed heuristics which perform symbolic analyses of formulae [3], [7]. Recently, however, several machine-learning based techniques have been shown to outperform or achieve competitive performance when compared to traditional heuristics-based methods [8], [9]. At present, there has been a rising interest in adapting neural approaches to both tasks [10]–[12], but the highly structural nature of logical formulae has made such adaptations challenging.

As such, neural methods that are structure-aware have outperformed their more basic counterparts, with the two most effective neural architectures being Tree LSTMs [13] and Graph Neural Networks (GNNs) [14]. However, in prior work Tree LSTMs and GNNs have been limited in their ability to represent global structure. That is, when Tree LSTMs have been used to embed the parse tree of a logical formula, they generate embeddings that represent the parse tree globally, but lose any notion of shared subexpressions and variable quantifications. With traditional GNN approaches, shared subexpressions and variable quantifications are captured, but the global graph embedding for a formula consists of a simple pooling operation over individual node embeddings that each represent only themselves and their local neighborhoods.

To address these two limitations, we present a novel approach that combines and extends GNNs [14], [15] and Tree LSTMs [13] to represent the complex structure of logical formulae. In particular, we introduce a hybrid method that produces high-quality node embeddings with a Message-Passing GNN [15], and then pools them together in a structure-dependent way with a DAG LSTM. The DAG LSTM pooling mechanism we introduce here is an extension of the Tree LSTM to DAG-structured data that works by simply staging and batching node updates according to a topological sort of the input graph. Each hidden state of the DAG LSTM then corresponds to a pooled representation of the GNN-generated node embeddings for some subgraph of the input formula.

Thus, our system overcomes the difficulties of prior GNNs-based methods in embedding more than just the local neighborhoods of nodes by using a LSTM-based method that computes the embedding for a node to be a function of its complete set of dependencies in the directed acyclic graph (DAG) form of its associated formula. Moreover, it overcomes the limitations of Tree LSTMs by more effectively representing global graph-structure through the reification of duplicate subexpressions and variables, thereby keeping the relevant contextual information surrounding each node (e.g. a variable with all the terms it appears in) in close proximity to that node (rather than in distinct regions of a tree-based representation).

We evaluate the binary classification performance of our models on the premise selection task with the Mizar dataset.
and on the proof step classification task with the HolStep dataset \cite{11}. The experiments show that our hybrid architecture outperforms all other neural approaches on both datasets.


ture outperforms all other neural approaches on both datasets. The experiments show that our hybrid architecture outperforms all other neural approaches on both datasets. The experiments show that our hybrid architecture outperforms all other neural approaches on both datasets. The experiments show that our hybrid architecture outperforms all other neural approaches on both datasets. The experiments show that our hybrid architecture outperforms all other neural approaches on both datasets. The experiments show that our hybrid architecture outperforms all other neural approaches on both datasets.

II. RELATED WORK

Premise selection is a binary classification task where the goal is to determine which premises (i.e. axioms of a specified theory) are necessary for proving a given conjecture (or theorem). Proof step classification is similar to premise selection in that it involves determining the relevance of a logical formula to a given conjecture, however, rather than classifying the axioms of some theory, classification instead involves proof steps (usually generated by a theorem prover during proof search). Proof steps differ from axioms in that they are generated through inferences involving combinations of several axioms, which makes them generally larger and hard to interpret for humans. When reasoning over large theories, e.g. the millions of axioms in Cyc \cite{4}, the ability to perform proof step classification and premise selection becomes critical, as the search space for proofs can explode within as little as one inference step.

We note that premise selection and proof step classification are not intrinsically machine learning tasks. The earliest approaches to premise selection \cite{3} were simple heuristics capturing the (transitive) co-occurrence of symbols in a given axiom and conjecture. Soon after, it was recognized that statistical machine-learning techniques would be effective tools for solving this problem. \cite{8} introduced a kernel method for premise selection where the similarity between two formulas was computed by the number of common subterms and symbols. The first deep learning approach to this problem was proposed by \cite{9}, who compared the performance of sequence models over character and symbol-level representations of logical formulae. In \cite{17}, the authors proposed a symbol-level method that learned low-dimensional distributed representations of function symbols and used those to construct concise embedded representations of given formulae that could be used for premise selection.

More recently, HolStep \cite{11}, a new formal dataset designed to be large enough to evaluate neural methods for premise selection and proof step classification (among other tasks), was introduced. Along with the dataset came a set of benchmark deep learning models that operated over character and symbol-level representations of higher-order logic formulae. FormulaNet \cite{18} was the first approach to transform a formula into a rooted-DAG (a modified version of the formula parse tree) and then embed the resulting graph with a GNN. Their paper included an elaborate series of experiments on the HolStep dataset showing the deficiencies of tree-based as compared to DAG-based representations of logical formulae.

The combined, online version of the aforementioned two tasks is referred to as proof guidance, and, as the name suggests, involves guiding the theorem prover towards exploring more productive regions of the proof search space for a given conjecture. Deep learning approaches to proof guidance include \cite{12}, where the authors explored a number of neural architectures in their implementation (including a structure-aware Tree LSTM that encoded the parse trees of logical formulae). Another approach \cite{19} represented formulae as DAGs with shared subexpressions and used Message-Passing GNNs to generate neural embeddings that could be used to guide theorem proving on a higher-order logic benchmark introduced in \cite{10}. However, like \cite{18}, the graph-level embeddings produced by their approach were simple, consisting of a max-pooling over the individual node embeddings of a given formula. This would cause any structure beyond local-neighborhood structure to not be considered in the final graph embedding.

Other conceptually-related works in this area include: 1) \cite{20} where the authors introduced a dataset for evaluating neural models on logical entailment and explored the use of several popular neural architectures (e.g. Tree LSTMs, BiDirLSTMs, etc.) on the proposed task. Their focus on logical entailment for propositional logic statements differs from our objectives regarding first-order and higher-order logic. 2) \cite{21} where the authors introduced the GamePad dataset (with baseline neural models) for evaluating neural models on the tasks of position evaluation (i.e. predicting how many remaining proof steps are needed to complete a proof) and tactic prediction (i.e. predicting what the next proof step will be).

III. FORMULA REPRESENTATION

A. Background

First-order logic formulae are formal expressions based on logical connectives and an alphabet of predicate, function, and variable symbols. A term is either a variable, a constant (functions with no arguments), or, inductively, a function applied to a tuple of terms. A formula is either a predicate applied to a tuple of terms or, inductively, a connective (e.g. ∧ reading as “and”) applied to some number of formulae. In addition, variables in formulae can be quantified by quantifiers (e.g. by ∃ reading as “for all”), where a quantifier introduces an additional semantic restriction for the interpretation of the variables it quantifies. Higher-order logics also allow for quantification over predicate and function symbols or the application of predicates over other predicates. For more details on both first-order and higher-order logic, we refer the reader to \cite{22}.

A graph is represented as a pair $G = (V, E)$, where $V = \{v_1, \ldots, v_n\}$ is the set of all vertices in $G$ and $E$ is the set of all edges in $G$, and each edge is a tuple of two vertices from $V$, i.e., of the form $e_{ij} = (v_i, v_j)$. The graphs here are rooted directed acyclic graphs, meaning that each edge $e_{ij} \in E$ is an ordered tuple, there is one vertex (the root) that never appears on the right-hand side of an edge, and there is no way of starting from any vertex $v$ in $V$ and traversing through edges in $E$ such that one returns to the original vertex $v$.

B. Logical Formulae as Graphs

The first question that arises when applying deep learning to automated reasoning regards representation. While the earliest
work on integrating deep learning with reasoning techniques used symbol- or word-level representations of input formulae [9], [11] (considering formula strings as words), subsequent work explored using formula parse trees [12], [20], [21] or rooted DAG forms [18], [19] as representation. When evaluated on the HolStep [11] and Holist [10] datasets, the DAG forms of logical formulae were found to be the more useful as compared to bag-of-symbols and tree-structured encodings [18], [19].

We focus on DAG representations of formulae; Figure 1 shows an example of such a representation. The DAG associated to a formula corresponds to its parse tree, where directed edges are added from parents to their arguments and shared subexpressions are mapped to the same subgraphs. We extend this transformation in two directions as follows. First, all instances of the same variable are collapsed into a single node (which maintains all prior connections), and the name of each variable is replaced by a generic variable token, as suggested in [18]. This simplifies the encoding and makes the representation name-invariant for variables (a well known property of logical formulae). Second, we collapse binarized formulae with a leading connective into a flatter form when doing so would not influence semantic meaning. For instance, our approach exploits the associativity and commutativity of \( \land \) and \( \lor \) to transform the formula \( (a \land (b \land (c \land d))) \) into the equivalent form \( \land (a, b, c, d) \). This results in a slightly more compact, but semantically equivalent graph. In practice, this did not influence our performance on validation data, however it yielded a slight increase in speed.

C. Edge Labeling

Capturing the ordering of arguments of logical expressions is still an open topic of research. [18] used a (self-named) tree-let encoding scheme that represents the position of a node relative to other arguments of the same parent as triples. On the other hand, [19] used positional edge labels, assigning to each edge a label which reflected the position of its target node in the argument list of the node’s parent (i.e. the source of the edge). We follow the latter strategy, albeit, with modifications.

In our formulation, the arguments to a particular node are given by a partial ordering. For logical connectives like conjunctions (\( \land \)) and biconditionals (\( \equiv \)), and predicates like equality (\( = \)), etc. all arguments are of the same rank. For other predicates, functions, and logical connectives like implications (\( \Rightarrow \)), the arguments are instead linearly ordered. However, we also support hybrid cases like simultaneous quantification over multiple variables. For instance, consider the formula \( \forall A, B, C, p(A, B, C) \). Each variable \( A, B, C \) is of the same rank (i.e. the variables could be reordered and the semantics would not change), but all variables must be ranked higher than the subexpression \( p(A, B, C) \). Thus, each of \( A, B, \) and \( C \) would have rank 1 and \( p \) would have rank 2.

The label given to each argument edge in the graph is the rank of the corresponding argument to the source parent node concatenated with the type of the parent node (\( \text{pred} \) and \( \text{func} \) for predicates and functions or the name of the connective or quantifier). In our example, the edge label between the \( \forall \) node and both \( A \) and \( B \) would be \( \forall_1 \), and between the \( \forall \) node and \( p \) would be \( \forall_2 \). Our approach also adds a parent edge in the reverse direction (i.e. from argument to parent) with a parent label. This is done to allow the message-passing phase of our method to accumulate relevant information about the many contexts in which each argument appears (i.e. a variable may appear as the argument to several terms).

To summarize, our work builds on the DAG conventions as in [18] and [19] but with two key differences in the graph representation: 1) commutative and associative leading connectives are collapsed into more compact, flattened forms; 2) edges are labeled with the rank of their attached argument node with respect to the partial ordering defined over their attached parent’s arguments.

IV. MODEL ARCHITECTURE

Our architecture combines two types of graph neural networks, first computing a set of initial node embeddings with a message-passing neural network (MPNN) and, then, passing those node embeddings into a DAG LSTM (which can be viewed as a sophisticated pooling mechanism). The DAG LSTM will accumulate information (beginning from the leaf nodes) upwards through the graph until the root node is reached, at which point it will terminate and return the final graph embedding, simply given by the hidden state of the root node of the input formula graph.

A. Graph Neural Network Node Embeddings

The MPNN framework can be thought of as an iterative update procedure that represents a node as an aggregation of information from its local neighborhood. Our approach leverages an MPNN to construct the set of initial node embeddings. To begin, our approach assigns each node \( v \) and edge \( e \) of the input graph \( G = (V, E) \) an initial embedding, \( x_v \) and \( x_e \). An initial node state is then computed by passing
each such embedding through feed-forward networks $F_V$ and $F_E$, producing node states $s_v^0 = F_V(x_v)$ and $h_v = F_E(x_v)$. Lastly, a message-passing phase runs for $t = 1, \ldots, T$ rounds, where each round consists of the following updates (note that, following existing work [15], [19], the edge embeddings are not updated between rounds):

$$m_v^t = \sum_{w \in N(v)} F_M^t([s_v^{t-1}; s_w^{t-1}; h_{e_{vw}}])$$

$$s_v^t = s_v^{t-1} + F_A^t([s_v^{t-1}; m_v^t])$$

where $F_M^t$ and $F_A^t$ are feed-forward neural networks unique to the $t$-th round of updates, and $[; ; ;]$ denotes vector concatenation. As in [15], $m_v^t$ should be considered the message to be passed to $s_v$, and $s_v^t$ represents the node embedding for node $v$ after $t$ rounds of iteration. Each of the node embeddings from the final round (i.e., $s_v^T$) will be passed to the subsequent pooling layer described in the next section.

B. DAG LSTM Subgraph Pooling

DAG LSTMs can be viewed as the extension of TreeLSTMs [13] to more general DAG-structured data. As with Tree LSTMs, DAG LSTMs compute each node embedding as the aggregated information of all their immediate predecessors. With initial node embeddings $s_v^T$ from the MPNN, node states $h_v$ are computed as follows.

$$i_v = \sigma(W_is_v^T + \sum_{w \in P(v)} U_i[h_w; h_{e_{vw}}] + b_i)$$

$$o_v = \sigma(W_os_v^T + \sum_{w \in P(v)} U_o[h_w; h_{e_{vw}}] + b_o)$$

$$c_v = \tanh(W_cs_v^T + \sum_{w \in P(v)} U_c[h_w; h_{e_{vw}}] + b_c)$$

$$f_vw = \sigma(W_fs_v^T + U_f[h_w; h_{e_{vw}}] + b_f)$$

$$c_v = i_v \odot c_v + \sum_{w \in P(v)} f_vw \odot c_w$$

$$h_v = o_v \odot \tanh(c_v)$$

where $\odot$ denotes element-wise multiplication, $\sigma$ is the sigmoid function, $P$ is a function that takes a node $v$ as an argument and returns the immediate children of $v$ in $G$, and $U_i, U_o, U_c,$ and $U_f$ are matrices. $i$ and $o$ represent input and output gates, while $c$ and $\hat{c}$ are intermediate computations (memory cells), and $f$ is a forget gate that modulates the flow of information from individual arguments into a node’s computed state. Each edge embedding $h_{e_{vw}}$ is generated as in the MPNN, but with a separate feed-forward network $F_E^t$ (not mentioned above).

In prior work [13], [23], there would be matrices for each edge type, (e.g., instead of just $U_i$, there would be $U_i^k$ for each edge type $k$). In this work, we opt for the simpler representation presented above to reduce the number of parameters in our already large model.

Importantly, node updates are computed in the order given by a topological sort of the graph, starting from the leaves (otherwise, one would risk computing a parent embedding before computing the embeddings of its children). Batching is done at the level of topological equivalence, i.e. every node with the same rank can have the updates computed simultaneously. Across batches of graphs, one could even do load-balancing to ensure that the number of nodes updated at each layer are roughly equivalent, since there are no dependencies between the nodes of graphs in different training examples.

C. Constructing Dependent Embeddings

Most prior neural approaches to both premise selection and proof guidance have opted to embed the premise and goal formulae independently of one another [9], [12], [18], [19]. They first embed the graph of the premise and then separately embed the graph of the goal, and finally learn a classifier (typically a deep feed-forward network) that determines whether the vector representations of both the premise and goal are similar enough to warrant exploration (for proof guidance) or entailment (for premise selection). In our view, this could be a limitation, as the contents of one formula seemingly should influence the embedding process of the other. We address this
in our work by exchanging information between the two graph embedding processes as follows.

Let \( S_P \) and \( S_C \) be the sets of node embeddings computed from the initial MPNN of our approach for the premise and conjecture graph. Each node embedding in both \( S_P \) and \( S_C \) is concatenated with the mean of embeddings for each identically labeled node from the opposite graph. Let \( \mathcal{I} \) be a function that takes a node and either \( S_P \) or \( S_C \) and returns all node embeddings from the set where the associated node has an identical label to the given node, i.e. \( \mathcal{I}(u, S_C) = \{ s_v \in S_C | u \equiv v \} \).

Our new node embeddings are then given by:

\[
\hat{S}_P = \left\{ \left[ s_u; \frac{1}{|\mathcal{I}(u, S_C)|} \sum_{s_v \in \mathcal{I}(u, S_C)} s_v \right] \mid s_u \in S_P \right\}
\]

\[
\hat{S}_C = \left\{ \left[ s_u; \frac{1}{|\mathcal{I}(u, S_P)|} \sum_{s_v \in \mathcal{I}(u, S_P)} s_v \right] \mid s_u \in S_C \right\}
\]

Figure 4 gives an example of this information exchange. Where the DAG LSTM would before use \( s_u \), it is replaced with its new form from \( \hat{S}_P \) or \( \hat{S}_C \). This mechanism allows information about the relative changes of nodes in one graph as compared to the other to influence what is aggregated during the LSTM pooling stage. We note that, because each node embedding is computed by the MPNN component, it is not necessarily the case that \( s_u = s_v \) (for \( u \equiv v \)) when the number of message-passing updates is greater than zero (as each node embedding accumulates information from its neighbors). Also, because variables have been anonymized in this point, we do not include them in this cross graph information process (doing so added too much noise), and instead simply concatenate them with a zero vector.

D. Final Prediction

For all experiments using only the MPNN component of our architecture, the inputs to the classifier network would be a max-pooling of the individual node embeddings for each graph, i.e. \( s_P = \text{POOL}(S_P) \) and \( s_C = \text{POOL}(S_C) \). When the LSTM was used, the final graph embeddings for the premise and conjecture were taken to be the hidden states for the root nodes of the premise and conjecture, \( s_P = h_{\text{root}}^P \) and \( s_C = h_{\text{root}}^C \). In either case, the resulting graph embeddings were concatenated and passed to a classifier feed-forward network \( F_{CL} \) for the final prediction \( F_{CL}(\{s_P; s_C\}) \).

V. Experiments

We evaluate our approach on the tasks of premise selection and proof step classification with two different datasets: HolStep\(^1\) and Mizar\(^2\). Consistent with prior work \([11, 17, 18]\), both tasks are treated as binary classification problems where, given a formula (either a premise or proof step) and a conjecture, our approach classifies the formula as either relevant or irrelevant to the conjecture.

A. Network Configurations and Training

In our model, all initial node embeddings were 128 dimensional vectors and all initial edge embeddings were 32 dimensional vectors. The initial feed-forward networks \( F_V \), \( F_E \), and \( F_E' \) expanded the embeddings to double their initial size and used a single hidden layer with dimensionality equal to the output size, i.e. 256 / 64. All other feed-forward networks (each \( F_M \), each \( F_A \), \( F_{CL} \)) followed mostly the same configuration, save for their input dimensionalities. Each consisted of two hidden layers with dimensionality equal to the output layer. Every hidden layer for all feed-forward networks would be followed by batch normalization \([24]\) and ReLU activation. The final activation for the classifier network \( F_{CL} \) was a sigmoid activation; for all other feed-forward networks, the final activations were ReLUs. In the LSTM, the hidden states were 320-dimensional vectors. For the vanilla DAG LSTM without cross-graph information, each of \( W_i, W_o, W_f, \text{ and } W_c \) were learned \( 256 \times 320 \) matrics. For the Graph LSTM with cross-graph information, each of \( W_i, W_o, W_f \), and \( W_c \) were learned \( 512 \times 320 \) matrics. In both LSTM versions, the

---

\(^1\)http://cl-informatik.uibk.ac.at/cek/holstep/

\(^2\)https://github.com/JUrban/deepmath
hidden states were of the same dimensionality, and thus each of $U$, $U_o$, $U_c$, and $U_f$ were learned $320 \times 320$ matrices.

Our models were constructed in PyTorch [25] and trained with the Adam Optimizer [26] with default settings. The loss function optimized for was binary cross-entropy. We trained each model for a maximum of 5 epochs on both HolStep and Mizar, as experiments over validation data found no gain from further training. Performance on the validation sets was evaluated after each epoch and the final model used for evaluation on the test data was taken from the best performing epoch at validation.

B. Datasets

1) Mizar Dataset: Mizar is a corpus of 57,917 theorems. Like [9] and [17], we use only the subset of 32,524 theorems which have an associated ATP proof (i.e. from an automated theorem prover), as those have been paired with both positive and negative premises (i.e. premises that do / do not entail a particular theorem) that can be used to train our approach. Following [9], we randomly split the 32,524 theorems: 80% / 10% / 10% for training, development, and testing. From these theorem files we then created 522,520 examples (i.e. 417,763 / 51,877 / 52,880 for train / development / test). An example consists of a premise, a conjecture, and a label for whether that premise is known to directly contribute to proving the conjecture.

We chose the Mizar dataset to evaluate our approach because of the adversarial method by which it was constructed. While the positive examples are simply those premises that were needed for the proof of their associated conjectures, the negative examples (i.e. premise-conjecture pairs where the premise was not needed to prove the conjecture) are formulae that were highly ranked by a k-NN premise selection approach trained on the complete dataset. This makes the premise selection problem much more difficult, as approaches relying on local structural differences or bag-of-words similarity have much less to work with. For example, consider Figures 5 and 6. In both cases, the premise and conjecture formulae are practically identical, with the only differences being the result of two label changes.

2) HolStep Dataset: HolStep is a large corpus designed to test machine learning approaches to automated reasoning. While the corpus provides training and evaluation data for a number of tasks in the automated reasoning domain, we will focus only on the portion needed for proof step classification. That part has 9,999 conjectures for training and 1,411 conjectures for testing, where each conjecture is paired with an equal number of positive and negative proof steps (i.e. proof steps that were / were not part of the final proof for the associated conjecture). From those 11,400 conjectures, 2,013,046 training examples and 196,030 testing examples are constructed. Each example is of the same form as used for Mizar. Positive examples were taken from the proof tree of a given conjecture, while negative examples were selected from the proof steps taken during the proof process that ultimately did not lead to a successful proof. We partitioned 10% of the training examples as a development set.

C. Baselines

For premise selection on Mizar, we experiment with two existing systems: FormulaNet (the code for testing FormulaNet on Mizar was taken from its associated github repository [3]) and the distributed formula representation of [17]. For the proof step classification task on Holstep, we compare against 4 systems implemented in two prior works: 1) DeepWalk and FormulaNet, both of which were applied to Holstep in [18]. 2) CNN-LSTM and CNN, both of which were introduced in the original Holstep paper [11]. For both datasets, we note that the MPNN by itself (not our hybrid system) can be considered another baseline (as an MPNN has been used for embedding formulae in [19]).

D. Experimental Results

In what follows, MPNN (Section IV-A) represents the performance of using only the message passing component of our architecture, MPNN-DagLSTM (Section IV-B) denotes the architecture that uses both the MPNN and DAG LSTM but without exchanging information between premise and conjecture embeddings, and Dep-MPNN-DagLSTM (Section IV-C) denotes the architecture that incorporates the cross-graph information (shown in Figure 4). $T$ indicates the number of MPNN update rounds. We note that when $T = 0$, the MPNN performs no message-passing updates. Thus, for $T = 0$, the MPNN outputs embeddings passed through only the initial feed-forward network $F_V$, e.g. for a node $v$ the MPNN component would return $s_v^0 = F_V(x_v)$.

https://github.com/princeton-vl/FormulaNet/
1) Mizar Dataset: Table I shows the performance of our approach, the basic MPNN architecture, and FormulaNet [18] for premise selection. The Dep-MPNN-DagLSTM architecture provides substantially better performance over FormulaNet, MPNN-DagLSTM and MPNN architectures with a 2.7% gain over the best alternative, which is statistically significant (z-test: $z = -10.4$; $p < .01$). We attribute the large gap to the adversarial strategy by which the dataset was constructed. The negative examples for premise selection (unlike proof step classification) involve real premises for other conjectures; thus, it is less likely that there would be features of the premise or conjecture that one could observe in isolation (e.g. too many symbols in a formula) that would make an entailment obvious. This makes strategies that embed the premise and conjecture independently less effective, as evidenced by our results.

We do not include the results of [17] in Table I as they partitioned their data such that test conjectures were mixed into the training data (we discuss the performance effects of this in detail in the following section). Additionally, the best model of [17] was seeded with node embeddings generated by a learning algorithm trained on all the Mizar examples (both training and testing data). We note, however, that in their experiments they achieved a performance of 76.45% after 2500 epochs of training. In a similar setting (see Figure 7), where we combined our training and development data and then re-partitioned it into a separate training and development set according to their partitioning methodology, our approach was able to obtain a performance of 77.6% on the new development set (without pre-training on test data).

2) HolStep Dataset: Table II shows the performance of our approach, the basic MPNN architecture, and the approaches proposed in [11] (CNN-LSTM, CNN) and [18] (Deepwalk, FormulaNet) on the task of proof step classification. Both Dep-MPNN-DagLSTM and MPNN-DagLSTM architectures for $T = 1$ and $T = 2$ outperforms FormulaNet to a statistically significant degree (with $p < .05$ for performance at 90.5% and $p < .01$ for all performance at and above 90.6%).

As a sanity check, we ran one additional experiment on HolStep to validate the difference between the DAG LSTM and MPNN methods. As the DAG LSTM pooling is somewhat similar to the MPNN without parent edges included (i.e. all nodes accumulate the information of their direct predecessors), we implemented this distinct system and compared its performance on the validation set with the LSTM approach. In particular, the MPNN was implemented with $T = 1$ and the exact same specifications and hyperparameters as were used in other experiments, except that no parent edges were included in the graph (i.e. nodes could only accumulate information from their immediate predecessors and not their parents). The DAG LSTM compared against was that which was used from the main experiments with $T = 0$ (because using the same values for $T$ would give the LSTM more information than was available to the MPNN). The ablated MPNN had a best validation performance of 88.5%, which was noticeably worse than both the MPNN with parent edges used in the main experiments (best validation performance of 90.1%) and the DAG LSTM (best validation performance of 90.0%). This clearly shows the distinction of the pooling mechanism against a system with access to the same information (nodes in relation to their immediate predecessors).

### Table I

| Model                  | T  | Accuracy |
|------------------------|----|----------|
| FORMULANET (2017)      | 3  | 71.4%    |
| MPNN                   |    |          |
|                        | 0  | 72.2%    |
|                        | 1  | 72.4%    |
|                        | 2  | 72.7%    |
| MPNN-DagLSTM           |    |          |
|                        | 0  | 72.8%    |
|                        | 1  | 72.4%    |
|                        | 2  | 72.2%    |
| DEP-MPNN-DagLSTM       |    |          |
|                        | 0  | 75.5%    |
|                        | 1  | 74.7%    |
|                        | 2  | 74.8%    |

### Table II

| Model                  | T  | Accuracy |
|------------------------|----|----------|
| DEEPWALK (2014)        | -  | 61.8%    |
| CNN-LSTM (2017)        | -  | 83.0%    |
| CNN (2017)             | -  | 82.0%    |
| FORMULANET (2017)      | 3  | 90.3%    |
| MPNN                   |    |          |
|                        | 0  | 77.2%    |
|                        | 1  | 89.7%    |
|                        | 2  | 90.0%    |
| MPNN-DagLSTM           |    |          |
|                        | 0  | 89.5%    |
|                        | 1  | 90.7%    |
|                        | 2  | 90.9%    |
| DEP-MPNN-DagLSTM       |    |          |
|                        | 0  | 89.6%    |
|                        | 1  | 90.6%    |
|                        | 2  | 90.5%    |

### VI. Discussion

#### A. Effect of Dataset Partitioning Scheme

For the Mizar experiments, we opted not to follow the dataset partitioning methodology of [17] where, rather than splitting on the theorem files, they split on the examples (i.e. they gathered all 522,520 examples and randomly partitioned those into train / development / test sets). This is a subtle but important distinction, as a conjecture from testing may be found in the training set.

To give a better idea as to why the distinction between splitting on the theorems versus splitting on the examples matters, we refer the reader to Figure 7. Both curves show how performance of the Dep-MPNN-DagLSTM approach (with $T = 0$ message-passing rounds) on the development set (y-axis) changes over the number of epochs (x-axis); but one line shows performance when the development set is partitioned as in [9] (i.e. partitioned on conjectures and not on examples)
while the other shows performance when the development set is partitioned as in [17] (i.e. partitioned on examples and not on conjectures). As might be anticipated, when test conjectures are mixed in with the training data, over-fitting becomes practically a non-issue and the problem becomes substantially easier.

B. Performance on Mizar vs. HolStep

Though the Dep-MPNN-DagLSTM architecture beat all baselines (in a statistically significant way) for both datasets, it was surprising that its performance on Holstep lagged behind that of the MPNN-DagLSTM. We posit that this difference resulted from the means by which both datasets were constructed.

In Mizar, examples were adversarially constructed with a k-NN classifier designed to fool structural / symbol similarity algorithms, while in Holstep, examples were taken from the outputs of a theorem prover. We believe this led to fundamentally different requirements between systems for both datasets. Namely, Mizar examples were dependent (looking at the premise without the conjecture, one could not tell if it was negative or positive) and HolStep examples were independent (there were properties of an individual proof step without the conjecture that would give away the positive or negative classification). This is partially supported by both [18] and [11], who noted similar findings for Holstep. Both works found their architectures to function just as well when performing classification on only the premise, rather than both on the premise and conjecture (90.0% versus 90.3% for FormulaNet and 83.0% versus 83.0% for CNN-LSTM).

Our hypothesis is that each system underperformed on its non-ideal dataset (i.e. MPNN-DagLSTM on Mizar and Dep-MPNN-DagLSTM on HolStep) because it had additional model complexity where it wasn’t needed. This could have caused overfitting to features that were not helpful (i.e. MPNN-DagLSTM would overfit to features useful for independent classification and Dep-MPNN-DagLSTM would overfit to features useful for dependent classification). That being said, we reiterate that the Dep-MPNN-DagLSTM was superior to all prior approaches in a statistically significant way across both datasets.

VII. CONCLUSIONS AND FUTURE WORK

In this work, we presented a novel method for computing vector-space embeddings of logical formulae that leverages their underlying DAG structure, thus addressing the limitations of embedding methods based on Tree LSTMs and GNNs. The next immediate plan for this work is to incorporate it directly into the proof search process of a theorem prover. By representing the entire state of a theorem prover as a single large graph (where each formula is connected to a single top-level node) one could use this technique to produce a single embedding for an entire theory that would potentially be useful for proof guidance. Another interesting future line of research would be incorporating more structural constraints into the embedding process. The Dep-MPNN-DagLSTM architecture let information flow through a many-to-many mapping between nodes of the two graphs. More compact mappings derived from not just the label similarity of nodes, but also the structural similarities (perhaps derived from works like [27]) could yield efficiency and performance gains.

REFERENCES

[1] G. Sutcliffe, “The tptp problem library and associated infrastructure,” Journal of Automated Reasoning, vol. 43, no. 4, p. 337, 2009.
[2] D. Ramachandran, P. Reagan, and K. Goolsby, “First-ordered researchcyc: Expressivity and efficiency in a common-sense ontology,” in AAAI workshop on contexts and ontologies: theory, practice and applications, 2005.
[3] K. Hoder and A. Voronkov, “Sine qua non for large theory reasoning,” in International Conference on Automated Deduction. Springer, 2011, pp. 299–314.
[4] C. Matuszek, M. Withbrock, J. Cabral, and J. DeOliveira, “An introduction to the syntax and content of cyc,” UMBC Computer Science and Electrical Engineering Department Collection, 2006.
[5] C. Kaliszyk and J. Urban, “Mizar 40 for mizar 40,” J. Autom. Reasoning, vol. 55, no. 3, pp. 245–256, 2015. [Online]. Available: https://doi.org/10.1007/s10817-015-9330-8
[6] A. Pease, I. Niles, and J. Li, “The suggested upper merged ontology: A large ontology for the semantic web and its applications,” in Working notes of the AAAI-2002 workshop on ontologies and the semantic web, vol. 28, 2002, pp. 7–10.
[7] R. Sekar, I. Ramakrishnan, and A. Voronkov, “Term indexing, handbook of automated reasoning,” 2001.
[8] J. Alama, T. Heskes, D. Kühnwein, E. Tsvitsivadze, and J. Urban, “Premise selection for mathematics by corpus analysis and kernel methods,” Journal of Automated Reasoning, vol. 52, no. 2, pp. 191–213, 2014.
[9] G. Irving, C. Szegedy, A. A. Alemi, N. Een, F. Chollet, and J. Urban, “Deeppath-deep sequence models for premise selection,” in Advances in Neural Information Processing Systems, 2016, pp. 2235–2243.
[10] K. Bansal, S. M. Loos, M. N. Rabe, C. Szegedy, and S. Wilcox, “Holist: An environment for machine learning of higher-order theorem proving (extended version),” arXiv preprint arXiv:1904.03244, 2019.
[11] C. Kaliszyk, F. Chollet, and C. Szegedy, “Holstep: A machine learning dataset for higher-order logic theorem proving,” arXiv preprint arXiv:1703.00426, 2017.
[12] S. Loos, G. Irving, C. Szegedy, and C. Kaliszyk, “Deep network guided proof search,” arXiv preprint arXiv:1701.06972, 2017.
[13] K. S. Tai, R. Socher, and C. D. Manning, “Improved semantic representations from tree-structured long short-term memory networks,” arXiv preprint arXiv:1503.00675, 2015.
[14] T. N. Kipf and M. Welling, “Semi-supervised classification with graph convolutional networks,” arXiv preprint arXiv:1609.02907, 2016.
