Improving Graph Neural Network Representations of Logical Formulae with Subgraph Pooling

Maxwell Crouse
mvcrouse@u.northwestern.edu
Northwestern University

Ibrahim Abdelaziz
ibrahim.abdelaziz1@ibm.com
IBM Research

Cristina Cornelio
cor@zurich.ibm.com
IBM Research

Veronika Thost
veronika.thost@ibm.com
MIT-IBM AI Lab

Lingfei Wu
wuli@us.ibm.com
IBM Research

Kenneth Forbus
forbus@northwestern.edu
Northwestern University

Achille Fokoue
achille@us.ibm.com
IBM Research

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 growing interest in adapting structure-aware neural methods to work with the underlying graph representations of logical expressions. While more effective than character and token-level approaches, graph-based methods have often made representational trade-offs that limited their ability to capture key structural properties of their inputs. In this work we propose a novel approach for embedding logical formulae that is designed to overcome the representational limitations of prior approaches. Our architecture works for logics of different expressivity; e.g., first-order and higher-order logic. We evaluate our approach on two standard datasets and show that the proposed architecture achieves state-of-the-art performance on both premise selection and proof step classification.

KEYWORDS

Premise Selection, Theorem Proving, Graph Neural Networks

1 INTRODUCTION

Automated theorem proving studies the design of automated systems that reason over logical theories (collections of *axioms* that are formulae known to be true) to generate formal proofs of given conjectures. It has been a longstanding, active area of artificial intelligence research that has demonstrated utility in the design of operating systems [1; 2], distributed systems [3; 4], compilers [5; 6], microprocessor design [7], and in general mathematics [8].

KDD'20 Deep Learning on Graphs Workshop, August 2020, San Diego, California, USA

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

Classical automated theorem provers (ATPs) have historically been most useful for solving problems that require complex chains of reasoning steps to be executed over smaller sets of axioms (see TPTP [9] for examples). When faced with problems for which thousands to millions of axioms are provided (only a handful of which may be needed at a time), even state-of-the-art theorem provers have difficulty [10; 11]. This deficiency has become more evident in recent years, as large logical theories [12–14] have become more widely available. A natural way to scale ATPs to broader domains has been to design sophisticated mechanisms that allow them to determine which axioms or intermediate proof outputs merit exploration in the proof search process. These mechanisms thus prune 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 useful to prove a given conjecture is referred to as *premise selection*, while the task of classifying intermediate proof steps as being a part of a successful proof for a conjecture is referred to as *proof step classification*.

Initial approaches for solving these two tasks proposed heuristics based on simple symbol co-occurrences between formulae [11; 15; 16]. While effective, these methods were soon surpassed by machine-learning techniques which could automatically adjust themselves to the needs of particular domains [17; 18]. At present, there has been a rising interest in developing neural approaches for both premise selection and proof step classification [19–21]; however, the complex and structured nature of logical formulae has made this development challenging. Neural approaches that take into account a formula's structure (e.g., parse tree), have been shown to outperform their more basic counterparts which operate on only symbols [22; 23]. The two most commonly used structure-aware neural methods have been Tree LSTMs [24] and GNNs [25]. However, as they have been applied in this domain, these methods appear to be leaving out useful structural information.

When used to embed the parse tree of a logical formula, Tree LSTMs generate embeddings that represent the parse tree globally, but they miss logically important information like shared subexpressions and variable quantifications. Conversely, traditional GNN approaches appear to better capture shared subexpressions and variable quantifications; however, the global graph embedding they

produce for the whole formula consists of a simple pooling operation over individual node embeddings; each representing only themselves and their local neighborhoods. Additionally, most prior approaches have embedded the premise and conjecture formulae independently of each other [18; 21–23]. They first embed the graph of the premise and then separately embed the conjecture graph, resulting in the contents of one formula having no influence on the embedding of the other.

To address these issues, we present a novel, two-phase embedding approach that operates over the DAG representations of logical formulae and is designed with careful consideration to their particular properties. Our method first produces an initial set of high-quality embeddings for nodes that incorporates more than just their local neighborhoods. Then, it pools the embeddings together in a structure-dependent way to generate a single graph-level embedding. This decoupling provides a clear point at which information between formulae can be exchanged, which allows us to define an attention-based exchange mechanism that can regulate information flow between the concurrent formula embedding processes.

We demonstrate the effectiveness and generality of our approach by evaluating classification performance on two standard datasets that involve different logical formalisms; the Mizar dataset [13; 18] for first-order logic and the Holstep dataset [20] for higher-order logic. Our experiments show the approach of this paper outperforms all previous approaches on the binary classification tasks of premise selection and proof step classification for both datasets. We also demonstrate how to easily integrate our approach with *E*, a well-established theorem prover [26], as its premise selection mechanism, allowing it to find more proofs (61.6% improvement) in a large-theory setting.

To summarize, our main contributions are: **1)** We show how to leverage the DAG structure implicit in logical formulae to produce more effective embeddings than traditional approaches operating over the local neighborhoods of individual nodes; **2)** We introduce a novel neural architecture that employs a localized attention mechanism to allow formulae to exchange information during the embedding process; **3)** We provide an extensive series of experiments and compare a range of neural architectures, showing significant improvement over existing state-of-the-art methods on two standard ATP classification datasets.

2 RELATED WORK

We note that premise selection and proof step classification are not intrinsically machine learning tasks. The earliest approaches to premise selection [11] were simple heuristics capturing the (transitive) co-occurrence of symbols in a given axiom and conjecture. Soon after, it was recognized that machine-learning techniques would be effective tools for solving this problem. The work of [17] introduced a kernel method for premise selection where the similarity between two formulae was computed by the number of common subterms and symbols. Deepmath [18] was the first deep learning approach to this problem, comparing the performance of sequence models over character and symbol-level representations of logical formulae. In [27], the authors proposed a symbol-level method that learned low-dimensional distributed representations of function symbols and used those to construct embedded representations of given formulae for premise selection. The work of [28] introduced a GNN for representing specifically first-order logic formulae in conjuctive normal form that captured certain logical invariances like reorderings of clauses and literals.

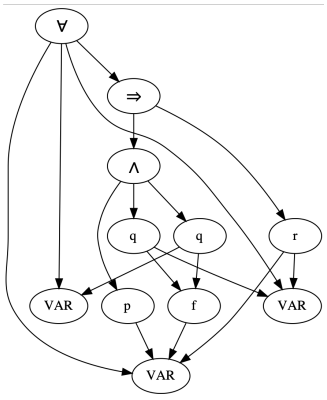

Figure 1: Shared subexpression graph representation

Recently, Holstep [20], 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 [22] was the first approach to transform a formula into a rooted DAG (a modified version of the parse tree) and then process the resulting graph with a GNN. Their GNN produced embeddings for each node within a formula's graph representation and then max pooled across node embeddings to get a formula-level embedding.

There are several other related works in this area that focus on different tasks (e.g., proof guidance, the combined, online version of the aforementioned two tasks). Deep learning approaches to proof guidance include [21], where the authors explored a number of neural architectures in their implementation (including a Tree LSTM that encoded parse trees of logical formulae). [23] represented formulae as DAGs with shared subexpressions and used message-passing GNNs (MPNNs) to generate embeddings that could be used to guide theorem proving on the higher-order logic benchmark of [19]. However, like [22], the graph-level embeddings produced by their approach were simple, consisting only of a max pooling over individual node embeddings. The work of [29] introduced a dataset for evaluating neural models on entailment for propositional logic and explored the use of several popular neural architectures on the proposed task. [30] where the authors introduced the GamePad dataset for evaluating neural models on the tasks of position evaluation and tactic prediction.

3 FORMULA REPRESENTATION

3.1 Logical Formulae as Graphs

While the earliest work on integrating deep learning with reasoning techniques used symbol- or word-level representations of input formulae [18; 20] (considering formula strings as words), subsequent work explored using formula parse trees [21; 29; 30] or rooted DAG

forms [22; 23]. On the Holstep [20] and Holist [19] datasets, the DAG forms of logical formulae were found to be the more useful than bag-of-symbols and tree-structured encodings [22; 23]. We focus on rooted-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. As in [22], all instances of the same variable are collapsed into a single node and the name of each variable is replaced by a generic variable token.

3.2 Edge Labeling

Capturing the ordering of arguments of logical expressions is still an open topic of research. [22] used a so-called *treelet* encoding scheme that represents the position of a node relative to other arguments of the same parent as triples. [23] used positional edge labels, assigning to each edge a label reflecting the position of its target node in the argument list of the node's parent. We follow the latter strategy, albeit, with modifications. In our formulation, edge labels are determined by a *partial ordering*. For unordered logical connectives (e.g., $\wedge$, $\Leftrightarrow$) and predicates (e.g., $=$) all arguments are of the same rank. For other predicates, functions, and logical connectives the arguments are instead linearly ordered. However, we also support hybrid cases like simultaneous quantification over multiple variables. The label given to each argument edge in the graph is the rank of the corresponding argument with respect to the parent concatenated with the type of the parent.

4 OUR APPROACH

In this work, we broadly distinguish between node embedding methods by *reachability*. More formally, consider a binary adjacency relation $\mathcal{R}$ defined for a set of graphs $\mathcal{G}$. The k-reachability relation $\mathcal{R}^k$ is given as the k-th power of $\mathcal{R}$, which is defined recursively with $\mathcal{R}^k = \mathcal{R}^{k-1} \circ \mathcal{R}$ and $\mathcal{R}^1 = \mathcal{R}$. We can define the transitive closure of $\mathcal{R}$ as simply $\mathcal{R}^+ = \mathcal{R}^\infty$. Letting the set of all nodes be $\bar{V} = \bigcup_{G=(V,E)\in\mathcal{G}} V$, we say that a graph embedding function f is a $\mathcal{R}^k$ embedding method if there exists a $k \in \mathbb{N}$ such that $\mathcal{R}^k \neq \mathcal{R}^+$ where for every u in $\bar{V}$ we have that f computes the value of u as a function of *only* the embeddings for $\{v \in \bar{V} \mid \mathcal{R}^k(u,v)\}$. Naturally, we define a $\mathcal{R}^+$ embedding method as one for which the opposite holds, i.e. for each u we have that f computes the value of u as a function of the embeddings for all v where $\mathcal{R}^+(u,v)$ holds. This distinction is particularly useful to make for graphs in the logic domain, as the transitive closure of adjacency is necessary for many key logical operations. As a trivial but important example, consider checking for the resolvability or unifiability of two ground formulas. Potentially all nodes of the two formulas would need to be examined, meaning that if both formulas had depth $> k$ then a procedure defined with $\mathcal{R}^k$ that checks only some subset of nodes within a fixed range of the root would be insufficient.

We view $\mathcal{R}^+$ embedding methods as those that perform a sophisticated type of subgraph pooling. That is, a $\mathcal{R}^+$ node embedding method computes the embedding for a node u as a function of the embeddings of all nodes reachable from u, i.e. a pooling of all such node embeddings. By definition, they incorporate as much graph context as is possible (i.e., the transitive closure of $\mathcal{R}$). While $\mathcal{R}^+$ node embedding methods naturally lend themselves to graph-level readout functions (and we will also use them in this way), we note that these concepts are defined for node-level embeddings (an important distinction to make, as for certain applications the input graphs could be disconnected).

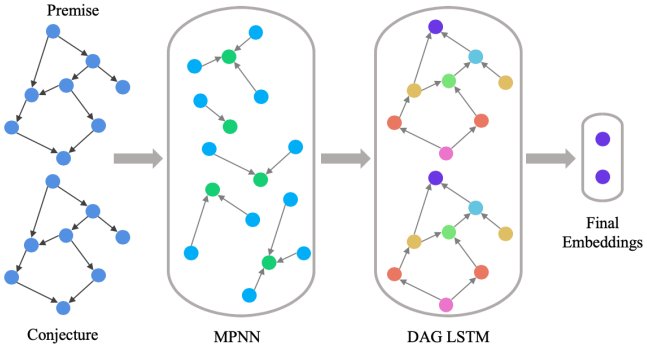

Figure 2: The overall embedding process with an MPNN initial node embedder and DAG LSTM pooling method.

Our approach operates in two stages (see Figure 2). First, a neural network generates embeddings for each node of an input formula's graph representation. Then, the node embeddings are passed into a follow-up $\mathcal{R}^+$ embedding method, referred to as the pooling method, that has $\mathcal{R}$ as the parent relation. The embedding for the root node of the input formula is returned, which is a function of all nodes in its graph. Our approach is very modular, with any node-level embedding method capable of serving as the initial node embedder (though we are mainly interested in $\mathcal{R}^+$ embedding methods) and any $\mathcal{R}^+$ embedding method being usable as the pooling method. Thus, in the next sections we describe the node embedding methods independently, and then we describe the classification process.

4.1 $\mathcal{R}^k$ Embedding Methods

4.1.1 Message-Passing Graph Neural Networks: 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. To begin, our MPNN assigns each node v and edge e of the input graph $G = (V, E)$ an initial embedding, x_v and x_e. Then, following [22], initial node states are computed by passing each such embedding through batch normalization [31] and a ReLU, producing node states $h_v^{(0)} = F_V(x_v)$ and edge states $h_e = F_E(x_e)$. Lastly, a message-passing phase runs for $t = 1, \ldots, k$ rounds

$$m_{v_p}^{(t)} = \sum_{w \in \mathcal{P}_\mathcal{A}(v)} F_{M_A}^{(t)}\left([h_v^{(t-1)} || h_w^{(t-1)} || h_{e_{vw}}]\right)$$

$$m_{v_c}^{(t)} = \sum_{w \in \mathcal{P}_C(v)} F_{M_C}^{(t)}\left([h_v^{(t-1)} || h_w^{(t-1)} || h_{e_{vw}}]\right)$$

$$h_v^{(t)} = h_v^{(t-1)} + F_A^{(t)}\left([h_v^{(t-1)} || m_{v_p}^{(t)} || m_{v_c}^{(t)}]\right)$$

where $\mathcal{P}_\mathcal{A}$ and $\mathcal{P}_C$ are functions that take a node v and return the immediate ancestors / children of v in G, and $F_{M_A}^{(t)}$, $F_{M_C}^{(t)}$, and $F_A^{(t)}$ are feed-forward networks unique to the t-th round of updates, and $||$ denotes vector concatenation. The reachability relation $\mathcal{R}$ in this context is defined as $\mathcal{R}(u,v) = \mathcal{A}(u,v) \vee C(u,v)$ where $\mathcal{A}$

and C are relations that hold true for immediate ancestor and child relationships, respectively. Similar to [32], $m_{v_p}^{(t)}$ and $m_{v_c}^{(t)}$ should be considered the *messages* to be passed to h_v, and $h_v^{(t)}$ represents the node embedding for node v after t rounds of iteration.

4.1.2 Graph Convolutional Neural Networks: Like with our MPNNs, for our Graph Convolutional Networks (GCNs) [25], the reachability relation $\mathcal{R}$ is given as the undirected adjacency relation, i.e., for nodes u and v we have $\mathcal{R}(u,v) = \mathcal{A}(u,v) \vee C(u,v)$. First, each node $v \in V$ is associated with an embedding h_v. Then, for $t = 1, \ldots, k$ rounds, updated embeddings are computed as

$$h_v^{(t)} = \phi\big(W^{(t)}\big(\frac{h_v^{(t-1)}}{|\mathcal{N}(v)|} + \sum_{w \in \mathcal{N}(v)} \frac{h_w^{(t-1)}}{\sqrt{|\mathcal{N}(v)||\mathcal{N}(w)|}}\big)\big)$$

where ϕ is a non-linearity (in this work, a ReLU), $\mathcal{N}(u) = \mathcal{P}_{\mathcal{A}}(u) \cup \mathcal{P}_C(u)$, and $W^{(t)}$ is a learned matrix for the t-th round of updates.

4.2 $\mathcal{R}^+$ Embedding Methods

4.2.1 DAG LSTMs: DAG LSTMs can be viewed as the generalization of Tree LSTMs [24] to DAG-structured data. With initial node embeddings s_v, the DAG LSTM uses the same N-ary equations as the Tree LSTM to compute node states h_v

$$\begin{aligned}
i_v &= \sigma\big(W_i s_v + \sum_{w \in \mathcal{P}_{\mathcal{R}}(v)} U_i^{(e_{vw})} h_w + b_i\big) \\
o_v &= \sigma\big(W_o s_v + \sum_{w \in \mathcal{P}_{\mathcal{R}}(v)} U_o^{(e_{vw})} h_w + b_o\big) \\
f_{vw} &= \sigma\big(W_f s_v + U_f^{(e_{vw})} h_w + b_f\big) \\
c_v &= i_v \odot \hat{c}_v + \sum_{w \in \mathcal{P}_{\mathcal{R}}(v)} f_{vw} \odot c_w \\
\hat{c}_v &= \tanh\big(W_c s_v + \sum_{w \in \mathcal{P}_{\mathcal{R}}(v)} U_c^{(e_{vw})} h_w + b_c\big) \\
h_v &= o_v \odot \tanh\big(c_v\big)
\end{aligned}$$

where $\odot$ denotes element-wise multiplication, σ is the sigmoid function and $U_i^{(e_{vw})}$, $U_o^{(e_{vw})}$, $U_c^{(e_{vw})}$, and $U_f^{(e_{vw})}$ are learned matrices (different for each edge type). 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. $\mathcal{P}_{\mathcal{R}}$ is a predecessor function that returns the set of nodes for which $\mathcal{R}$ holds true, i.e. $\mathcal{P}_{\mathcal{R}}(u) = \{v \in V \mid \mathcal{R}(u,v)\}$. In this work, it returns either the parents or the children, depending on whether the direction of accumulation is desired to go upwards or downwards. For readability, we omitted the layer normalization [33] applied to each matrix multiplication (e.g., $W_i s_v$, $U_i h_w$, etc.) from the above equations. Each instance of layer normalization maintained its own separate parameters.

The DAG LSTM we propose here is nearly the same as the Tree LSTM of [24], however there are key implementational differences between the two approaches. In Tree LSTMs, $\mathcal{P}_{\mathcal{R}}$ typically returns child nodes (since a node can have only one parent), while in our work it can return either children or parents. In addition, batching together node updates in a Tree LSTM can be done at the level of depth (i.e., all nodes at the same depth in the tree can have their updates computed simultaneously); however, with DAGs this batching strategy could cause a node to be updated and overwritten multiple times. To solve this, we propose the use of *topological batching*. In our approach, node updates are computed in the order given by a topological sort of the graph, starting from the leaves (or root depending on $\mathcal{P}$), with updates batched together at the level of topological equivalence, i.e., every node with the same rank can have the updates computed simultaneously.

4.2.2 Attention-Enhanced DAG LSTMs: In order to allow the contents of the premise and conjecture to influence one another during the embedding process, we introduce a localized attention mechanism that exchanges information between the two graph embeddings. Let S_P and S_C be the sets of node embeddings computed by some initial node embedder for the premise and conjecture graphs. 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 approach computes multi-headed attention scores [34] between identically labeled nodes and uses those attention scores to build new embeddings that provide cross graph information to the pooling procedure. For an input u, for each $k_j \in \mathcal{I}(u, S_C)$ we compute

$$\hat{q}_i = W_i^{(q)} s_u, \quad k_{ij} = W_i^{(k)} k_j, \quad v_{ij} = W_i^{(v)} k_j$$

where $W_i^{(q)}$, $W_i^{(k)}$, and $W_i^{(v)}$ are learned matrices for each of the $i = 1, \ldots, N$ attention heads.

$$w_{ij} = \frac{\hat{q}_i^\top k_{ij}}{\sqrt{b_{\hat{q}}}}, \quad \alpha_{ij} = \frac{\exp(w_{ij})}{\sum_{j'} \exp(w_{ij'})},$$

where $b_{\hat{q}}$ is the dimensionality of $\hat{q}_i$ and α_{ij} is computed as

$$q_i = \sum_j \alpha_{ij} v_{ij}, \quad s'_u = \sigma(W^{(g)} r_u) \odot (W^{(o)} \Big\|_{i=1}^{N} q_i)$$

The final vector s'_u for input s_u is a combination of its N transformations, with $W^{(g)}$ and $W^{(o)}$ being learned matrices, r_u a learned vector for the type (e.g., quantifier, predicate, etc.) of node u, and $\|$ denoting vector concatenation over the N attention vectors. The gating mechanism $\sigma\big(W^{(g)} r_u\big)$ we propose here simply allows for the architecture to cut off information flow between the two graphs if doing so improves loss, thus turning the architecture into the simpler DAG LSTM introduced previously. Following the attention computation, each s_u is replaced by $\bar{s}_u = s_u || s'_u$ and a DAG LSTM then processes each node embedding.

4.2.3 Bidirectional DAG LSTMs: We also explore a simple extension of the DAG LSTM from above to a Bidirectional DAG LSTM. In this method, the embeddings for a node are computed with the node's complete set of ancestors and descendants, i.e. with respect to $\mathcal{A}^+ \cup C^+$. For a node u and graph G, the embedding s_u is

$$s_u = F_{BD}\big([\text{DAG-LSTM}^{\uparrow}(u, G) \,||\, \text{DAG-LSTM}^{\downarrow}(u, G)]\big)$$

where F_{BD} is a feed-forward network, and DAG-LSTM$^{\uparrow}$ / DAG-LSTM$^{\downarrow}$ are both DAG LSTMs (following the design presented before) which set $\mathcal{R}$ as $\mathcal{A}$ and C, respectively.

4.3 Classification Process

In our approach, the final graph embeddings for the premise and conjecture are taken to be the embeddings for the root nodes of the premise and conjecture, $s_P = h^P_{root}$ and $s_C = h^C_{root}$. For ablation experiments using *only* local neighborhood-based node embedders (MPNN / GCN from Section 4.1), the inputs to the classifier network would be a max pooling of the individual node embeddings for each graph. In either case, the two graph embeddings are concatenated and passed to a classifier feed-forward network F_{CL} for the final prediction $F_{CL}([s_P; s_C])$.

5 EXPERIMENTS AND RESULTS

In this section, we evaluate our approach to show 1) how accurately can it predict the label of an axiom or proof step and 2) an ablation study that shows the effect of different node embedding and pooling mechanisms. We compare our approach to prior works using two standard datasets: Mizar[1] [13] and Holstep[2] [20].

5.1 Datasets

5.1.1 Mizar Dataset: Mizar [13] is a corpus of 57,917 theorems. Like [18; 27; 28], we use only the subset of 32,524 theorems which have an associated ATP proof, as those have been paired with both positive and negative premises (i.e., axioms that do / do not entail a particular theorem) to train our approach. We randomly split the 32,524 theorems as 80% / 10% / 10% for training, development, and testing (yielding 417,763 / 51,877 / 52,880 individual premises). Following [28], each example given to the network consisted of a conjecture paired with the complete set of both positive and negative premises. The task was then to classify each individual premise as positive or negative.

5.1.2 Holstep Dataset: Holstep [20] is a large corpus designed to test machine learning approaches on automated reasoning. Following prior work [20; 22], we use only 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). Using that data, we obtain 2,013,046 training examples and 196,030 testing examples, where each example is a triple with the proof step, conjecture, and a positive or negative label. We held out 10% of the training set to be used as a development set. We follow the binary classification problem setting of [22] and [20].

5.2 Classification Experiments

5.2.1 Baselines: For premise selection on Mizar, we compare with two existing systems: the distributed formula representation of [27] and the property-invariant formula representation of [28]. For proof step classification on Holstep, we compare against 4 systems implemented in two prior works: 1) DeepWalk [35] and FormulaNet [22], both of which were applied to Holstep in [22]. 2) CNN-LSTM and CNN, both introduced with the Holstep dataset [20].

5.2.2 Main Results: Table 1 shows the performance for the version of our approach that incorporates the entire context surrounding a node into its embedding and jointly embeds premises / proof steps with the conjecture, i.e., a Bidirectional DAG LSTM with attention-enhanced DAG LSTM pooling. Overall, our system outperforms all state-of-the-art systems on both datasets using a standard evaluation on held-out test data. It outperforms by a large margin on Mizar (+4.5%, which is statistically significant with $p < 0.01$) and by a moderate, but still statistically significant, margin on Holstep (+1.1% with $p < 0.01$). In addition to the standard evaluation using a held-out test dataset reported on Table 1, we also compare to [28], which introduced a GNN designed to process specifically first-order logic theories in conjunctive normal form. In their evaluation on Mizar, they split their data into only a train and test set and evaluated the model obtained at each epoch on their test set, reporting an accuracy of "around 80%" as the best performance across all test set evaluations. Following their setup, our best validation performance is 81.9%, a roughly 2% gain over [28].

Table 1: Experimental results (accuracy) for Mizar and Holstep test, best result for both datasets in bold

Formula Embedding Method	Mizar	Holstep
Kucik & Korovin (2018) [27]	76.5%	–
DeepWalk (2014) [35]	–	61.8%
CNN-LSTM (2017) [20]	–	83.0%
CNN (2017) [20]	–	82.0%
FormulaNet (2017) [22]	–	90.3%
BidirDagLSTM-AttDagPool (this work)	**81.0**%	**91.4**%

Table 1 confirms our hypothesis that a more holistic treatment of logical formulae can result in a more effective embedding process than simpler methods that, by their implementation, consider less structure and embed premises and conjectures independently.

5.2.3 Ablation Studies. We present the results of our ablations in Table 2. On Mizar, we can see that variants with attention-based pooling were the most performant by a large margin. When controlling for pooling type, the $\mathcal{R}^+$ node embedders provided better performance than the $\mathcal{R}^k$ node embedders. Similarly, when controlling for node embedding type, $\mathcal{R}^+$ pooling methods provided improvement over max pooling.

For Holstep, when controlling for node embedding type the $\mathcal{R}^+$ pooling methods had better performance than max pooling. Interestingly, when controlling for pooling type, the difference between the MPNN and $\mathcal{R}^+$ node embedding methods was not significant. Within approaches introduced here, those variants using AttDagPool did not significantly improve over those using DagPool. We suspect that this is due to the fundamental difference between proof step classification and premise selection. Intermediate proof steps are typically much larger and noisier than actual premises, which may have led to Holstep example pairs being independent (i.e., there were properties of an individual proof step without the conjecture that would give away the positive or negative label). This is partially supported by both [22] and [20], who observed that their architectures performed just as well when classifying with only the proof step, rather than on both the proof step and conjecture (90.0% vs. 90.3% for FormulaNet and 83.0% vs. 83.0% for CNN-LSTM). On validation data, we also explored higher numbers

[1] https://github.com/JUrban/deepmath
[2] http://cl-informatik.uibk.ac.at/cek/holstep/

Table 2: Ablation study on Mizar and Holstep test sets.

Node Embedding	Pool Type	k	Mizar	Holstep
MPNN	MaxPool	2	76.9%	90.5%
MPNN	DagPool	2	77.4%	91.3%
MPNN	AttDagPool	2	79.7%	91.3%
GCN	MaxPool	2	74.7%	89.0%
GCN	DagPool	2	77.3%	90.9%
GCN	AttDagPool	2	79.8%	90.8%
DagLSTM	DagPool	–	78.4%	91.4%
DagLSTM	AttDagPool	–	80.7%	91.5%
BidirDagLSTM	DagPool	–	78.1%	91.4%
BidirDagLSTM	AttDagPool	–	81.0%	91.4%

of update rounds (i.e., the k parameter) for variants of our approach using an MPNN as the initial node embedder; however, like [22] we found insignificant change beyond $k = 2$.

5.3 Premise Selection for Automated Theorem Proving

To show that our approach could be used to improve the performance of an actual theorem prover, we ran a traditional premise selection experiment with E [26]. We first trained new models using our settings from the classification experiments, however, this time optimizing for binary classification between pairs of individual formulae. In addition to our Mizar training set from before, we also augmented our training data by adding randomly selected negative examples for each example from our original training set. For testing, we paired the conjecture of each of the 3,252 problems from our Mizar validation set with the complete set of statements from all chronologically preceding problems (as described in [18]) in the union of our training and validation sets. For each problem, our model then ranked the premises with respect to each conjecture and returned the top $k \in \{16, 32, 64, 128, 256, 512, 1024, 2048, \infty\}$ premises (where ∞ indicates including all premises).

E was run on each problem in *auto-schedule* mode (which tries several expert heuristics based on the given problem) with a time limit of 10 seconds per k, stopping at the first k where the problem was solved. To validate that our approach solves more problems than E would have by itself in the same amount of time, we also measured the performance of E when run with all premises (identical to $k = \infty$) for 90 seconds per problem. Out of 3,252 problems, E by itself was able to solve 918; however, using our approach as its premise selection mechanism, E was capable of solving 1484. In both settings, E had the same amount of time (90 seconds) per problem to find a proof, but with our approach it was able to solve 566 more problems (a 61.6% improvement) which is statistically significant with $p < 0.01$.

6 CONCLUSIONS AND FUTURE WORK

In this work, we introduced a novel method for computing neural representations of logical formulae that was designed with careful consideration to their unique structural properties. Our approach achieved new state-of-the-art performances on two standard datasets, despite the datasets involving different logical formalisms. We also showed how to easily incorporate our method with an existing theorem prover as its premise selection mechanism; where its inclusion led to a 61.6% improvement in terms of number of proofs found.

An interesting future line of research could be in exploring the attention mechanism introduced in Section 4.2.2. One could see if different pairing operations than identicality (e.g., structural similarities derived from works like [36]) lead to more efficiency with better performance. One might also explore the effect that the initial node embedder has on the similarities computed by the attention mechanism. When using an MPNN for node embeddings, one might expect the attention mechanism to be computing local neighborhood similarity; however, when using a DAG LSTM, it is less obvious what type of similarity is being captured. Inspecting what the neural network learns to be the most useful subgraphs to match could lead to insights that produce new heuristics for proof guidance.

REFERENCES

[1] Gerwin Klein. Operating system verification—an overview. *Sadhana*, 34(1):27–69, 2009.

[2] Gerwin Klein, June Andronick, Kevin Elphinstone, Toby Murray, Thomas Sewell, Rafal Kolanski, and Gernot Heiser. Comprehensive formal verification of an os microkernel. *ACM Transactions on Computer Systems (TOCS)*, 32(1):1–70, 2014.

[3] Stephen J Garland and Nancy A Lynch. The ioa language and toolset: Support for designing, analyzing, and building distributed systems. Technical report, Technical Report MIT/LCS/TR-762, Laboratory for Computer Science . . . , 1998.

[4] Chris Hawblitzel, Jon Howell, Manos Kapritsos, Jacob R Lorch, Bryan Parno, Michael L Roberts, Srinath Setty, and Brian Zill. Ironfleet: proving practical distributed systems correct. In *Proceedings of the 25th Symposium on Operating Systems Principles*, pages 1–17, 2015.

[5] Paul Curzon and P Curzon. A verified compiler for a structured assembly language. In *TPHOLs*, pages 253–262, 1991.

[6] Xavier Leroy. Formal verification of a realistic compiler. *Communications of the ACM*, 52(7):107–115, 2009.

[7] Warren A Hunt. Microprocessor design verification. *Journal of Automated Reasoning*, 5(4):429–460, 1989.

[8] Thomas Hales, Mark Adams, Gertrud Bauer, Tat Dat Dang, John Harrison, Hoang Le Truong, Cezary Kaliszyk, Victor Magron, Sean McLaughlin, Tat Thang Nguyen, et al. A formal proof of the kepler conjecture. In *Forum of Mathematics, Pi*, volume 5. Cambridge University Press, 2017.

[9] Geoff Sutcliffe. The tptp problem library and associated infrastructure. *Journal of Automated Reasoning*, 43(4):337, 2009.

[10] Deepak Ramachandran, Pace Reagan, and Keith Goolsbey. First-orderized researchcyc: Expressivity and efficiency in a common-sense ontology. In *AAAI workshop on contexts and ontologies: theory, practice and applications*, 2005.

[11] Kryštof Hoder and Andrei Voronkov. Sine qua non for large theory reasoning. In *International Conference on Automated Deduction*, pages 299–314. Springer, 2011.

[12] Cynthia Matuszek, Michael Witbrock, John Cabral, and John DeOliveira. An introduction to the syntax and content of cyc. *UMBC Computer Science and Electrical Engineering Department Collection*, 2006.

[13] Cezary Kaliszyk and Josef Urban. Mizar 40 for mizar 40. *J. Autom. Reasoning*, 55(3):245–256, 2015.

[14] Adam Pease, Ian Niles, and John 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*, volume 28, pages 7–10, 2002.

[15] Alex Roederer, Yury Puzis, and Geoff Sutcliffe. Divvy: An atp meta-system based on axiom relevance ordering. In *International Conference on Automated Deduction*, pages 157–162. Springer, 2009.

[16] Daniel Kühlwein, Twan van Laarhoven, Evgeni Tsivtsivadze, Josef Urban, and Tom Heskes. Overview and evaluation of premise selection techniques for large theory mathematics. In *International Joint Conference on Automated Reasoning*, pages 378–392. Springer, 2012.

[17] Jesse Alama, Tom Heskes, Daniel Kühlwein, Evgeni Tsivtsivadze, and Josef Urban. Premise selection for mathematics by corpus analysis and kernel methods. *Journal of Automated Reasoning*, 52(2):191–213, 2014.

[18] Geoffrey Irving, Christian Szegedy, Alexander A Alemi, Niklas Een, François Chollet, and Josef Urban. Deepmath-deep sequence models for premise selection. In *Advances in Neural Information Processing Systems*, pages 2235–2243, 2016.

[19] Kshitij Bansal, Sarah Loos, Markus Rabe, Christian Szegedy, and Stewart Wilcox. Holist: An environment for machine learning of higher order logic theorem proving. In *International Conference on Machine Learning*, pages 454–463, 2019.
[20] Cezary Kaliszyk, François Chollet, and Christian Szegedy. Holstep: A machine learning dataset for higher-order logic theorem proving. *International Conference on Learning Representations*, 2017.
[21] Sarah Loos, Geoffrey Irving, Christian Szegedy, and Cezary Kaliszyk. Deep network guided proof search. *International Conference on Logic for Programming, Artificial Intelligence, and Reasoning*, 2017.
[22] Mingzhe Wang, Yihe Tang, Jian Wang, and Jia Deng. Premise selection for theorem proving by deep graph embedding. In *Advances in Neural Information Processing Systems*, pages 2786–2796, 2017.
[23] Aditya Paliwal, Sarah Loos, Markus Rabe, Kshitij Bansal, and Christian Szegedy. Graph representations for higher-order logic and theorem proving. *Proceedings of AAAI 2020*, 2019.
[24] Kai Sheng Tai, Richard Socher, and Christopher D Manning. Improved semantic representations from tree-structured long short-term memory networks. In *Proceedings of the 53rd Annual Meeting of the Association for Computational Linguistics and the 7th International Joint Conference on Natural Language Processing (Volume 1: Long Papers)*, pages 1556–1566, 2015.
[25] Thomas N Kipf and Max Welling. Semi-supervised classification with graph convolutional networks. *International Conference on Learning Representations*, 2016.
[26] Stephan Schulz. System description: E 1.8. In *International Conference on Logic for Programming Artificial Intelligence and Reasoning*, pages 735–743. Springer, 2013.
[27] Andrzej Stanisław Kucik and Konstantin Korovin. Premise selection with neural networks and distributed representation of features. *arXiv preprint arXiv:1807.10268*, 2018.
[28] Miroslav Olšák, Cezary Kaliszyk, and Josef Urban. Property invariant embedding for automated reasoning. *arXiv preprint arXiv:1911.12073*, 2019.
[29] Richard Evans, David Saxton, David Amos, Pushmeet Kohli, and Edward Grefenstette. Can neural networks understand logical entailment? *International Conference on Learning Representations*, 2018.
[30] Daniel Huang, Prafulla Dhariwal, Dawn Song, and Ilya Sutskever. Gamepad: A learning environment for theorem proving. *International Conference on Learning Representations*, 2019.
[31] Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. In *International Conference on Machine Learning*, pages 448–456, 2015.
[32] Justin Gilmer, Samuel S Schoenholz, Patrick F Riley, Oriol Vinyals, and George E Dahl. Neural message passing for quantum chemistry. In *Proceedings of the 34th International Conference on Machine Learning-Volume 70*, pages 1263–1272. JMLR. org, 2017.
[33] Jimmy Lei Ba, Jamie Ryan Kiros, and Geoffrey E Hinton. Layer normalization. *arXiv preprint arXiv:1607.06450*, 2016.
[34] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. In *Advances in neural information processing systems*, pages 5998–6008, 2017.
[35] Bryan Perozzi, Rami Al-Rfou, and Steven Skiena. Deepwalk: Online learning of social representations. In *Proceedings of the 20th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 701–710. ACM, 2014.
[36] Brian Falkenhainer, Kenneth D Forbus, and Dedre Gentner. The structure-mapping engine: Algorithm and examples. *Artificial intelligence*, 41(1):1–63, 1989.
[37] Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer. Automatic differentiation in pytorch. 2017.
[38] Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

7 APPENDIX

7.1 Network Configurations

For Holstep, our hyperparameters were chosen to be comparable to [22]. In our model, node embeddings were 256-dimensional vectors and edge embeddings were 64-dimensional vectors. All feed-forward networks (each $F^{(t)}_{M_A}$, each $F^{(t)}_{M_C}$, each $F^{(t)}_{A}$, F_{BD}, and F_{CL}) followed mostly the same configuration, except for their input dimensionalities. Each had one hidden layer with dimensionality equal to the output layer (except for F_{CL} where the dimensionality was half the input dimensionality). Every hidden layer for all feed-forward networks (except for F_{CL}) was followed by batch normalization [31] and a ReLU. The final activation for F_{CL} was a sigmoid; for all other feed-forward networks, the final activations were ReLUs. For the DAG LSTMs, the hidden states were 256-dimensional vectors. Each $U_i^{(e_{vw})}$, $U_o^{(e_{vw})}$, $U_c^{(e_{vw})}$, and $U_f^{(e_{vw})}$ were learned 256×256 matrices and each of W_i, W_o, W_f, W_c, W_a, and W_g were learned 256×256 matrices. For Mizar, all above dimensionalities were halved to be comparable to [27; 28]. For the attention-enhanced DAG LSTM, the multi-headed attention mechanism used two heads, with each $W_i^{(q)}$, $W_i^{(v)}$, and $W_i^{(k)}$ mapping from the node state dimensionality to double the node state dimensionality.

7.2 Training

Our models were constructed in PyTorch [37] and trained with the Adam Optimizer [38] with default settings. The loss function optimized for was binary cross-entropy. We trained each model for 5 epochs on Holstep and 30 epochs on Mizar, as validation performance did not improve with more training. Performance on the validation sets was evaluated after each epoch and the best performing model on validation was used for the single evaluation on the test data.

7.3 Hardware Setup

All experiments were run on Linux machines with 72-core Intel Xeon(R) 6140 CPUs @ 2.30 GHz and 750 GB RAM, and two Tesla P100 GPUs with 16 GB GPU memory.