Premise Selection for Automated Theorem Proving using Deep Neural Networks

Harsh Gupta
hgupta10@illinois.edu

Seo Taek Kong
skong10@illinois.edu

Siddhartha Satpathi
ssatpth2@illinois.edu

Abstract

In this project, we consider the problem of premise selection for automated theorem proving. We focus on the recent paper by Wang et al. (see [5]) and use graph-based embeddings to model mathematical statements as inputs to a neural network-based binary classification model. As we will see from experiments on the Holstep dataset, using graph-based embeddings captures the intrinsic structure in a mathematical statement naturally, leading to superior classification performance.

1. Introduction

Automated Theorem Proving (ATP) refers to using a set of computer programs to prove mathematical theorems. An ATP solver typically requires all the mathematical statements needed to prove a theorem, in order to generate the proof. A simple way to achieve this is to input all the known mathematical statements to the ATP solver. However, the set of all known mathematical statements can be prohibitively huge. Moreover, the complexity of ATP solvers typically scales exponentially with the number of input mathematical statements (see [1]). Hence, it is important to carefully curate the set of relevant statements (needed to prove a particular theorem), from the space of all mathematical knowledge, in order to provide the ATP with a small and relevant input of statements. This process of selecting mathematical statements that are relevant to prove a particular theorem is known as premise selection. In this report, we focus on an approach to carry out premise selection using graph-based embeddings, combined with deep learning. Note that we do not focus on the task of automated theorem proving in this report, and assume the ATP solver to be a black box. We focus on the premise selection task, which precedes the automated theorem proving task.

Let us consider an example to understand the idea behind premise selection. Suppose we want to prove the following conjecture: $C = \{\sqrt{2} \text{ is irrational }\}$. The ATP solver needs a set of relevant statements to be fed into it, in order to successfully generate the proof. Suppose we have a set of known mathematical statements given by

$$S_C = \{\text{two even numbers are relatively prime},\\ 5 \text{ is a prime number}, a + b = b + a\}$$

We want the premise selection algorithm to label the statements in S_C as "relevant" or "irrelevant" to proving the conjecture C. For human mathematicians, it is evident that only the first statement in the set S_C is relevant to proving the conjecture C, while the other two statements are not useful.

We aim to design a learning algorithm which is able to separate the set of relevant statements from the irrelevant ones, given a particular conjecture. We consider the following supervised learning approach: For each conjecture C, we have a set of statements S_C, with each statement labelled $+1$ if it is relevant in proving C, or labelled 0 otherwise. Subsequently, we train a classification model using training samples of $(conjecture, statements, labels)$. The classification model predicts the labels for a new conjecture C and a corresponding set of known mathematical statements S_C. We use a neural network-based binary classification model to perform this training.

An important question that arises in the modelling of the premise selection problem is the following: How to input a conjecture or a statement to the binary classification model? One naive way to do this would be to input the vanilla representation of a conjecture or a statement, i.e., input them as a sequence of mathematical symbols. But, the major disadvantage with the above methodology is that it is unable to capture the intrinsic mathematical structure in a statement. In this project, we will generate graph-based embeddings (similar to [5]) for the statements before feeding them into the binary classification model. The idea of using graph-based embeddings for statements is useful, as it naturally captures the flow of information within a mathematical statement, while simultaneously capturing structure. As we will see, use of graph-based embeddings significantly improves the performance of our model.

1.1. Related work

The paper [1] is among the very few papers which use neural networks for premise selection. Authors in [1] use a different data set than what we used (they use the Mizar

dataset [1], while we use the HolStep dataset [2]), so we will not be directly comparing their results with ours. Nevertheless, we qualitatively explain the differences in their approach and ours. Before feeding the conjecture-statement pairs to the binary classification neural network, authors in [1] extract features from the formulas[3]. This process of extracting features from the formulas is done through neural network-based sequence models. A conjecture and the corresponding set of statements are fed as a sequence to different neural networks, designed with recurrent-convolutional layers or LSTM-based layers. The output of these sequence models is subsequently fed into the classifier. This is known as character-level embedding. Authors in [1] also employ word-level embedding which basically captures the similarity of symbols across different statements.

Although the embedding process in [1] is intelligent, it treats mathematical statements as natural language sentences, which is non-ideal. It fails to capture the additional grammatical and recursive structure which is intrinsic to mathematical formulas. To remedy this, authors in [5] use graph-based embeddings to capture the structure in a mathematical statement, as opposed to working with the characters or words in the statement directly. In this project, we reproduce the results in [5], implementing graph-based embeddings to model the recursive structure in a mathematical formula. This process captures the grammatical and syntactical structure in a mathematical formula. Another advantage of this representation is that it can also be easily made invariant to the renaming of variables in a mathematical formula.

Let us explain the big-picture idea behind a graph-based embedding using a simple example. Suppose we have the following mathematical formula: $\forall x \exists y (P(x) \wedge Q(x, y))$. This can be naturally represented as a graph, as shown in Figure 1. The detailed procedure of this conversion is presented in Section 2. Note that this graph captures the flow of information within the formula. Also, the variables x and y are both given the same name VAR, so that the embedding is invariant to the variable names. Observe that each node in the graph corresponds to a symbol in the formula. To compute the representation of the graph, we first compute the graph-based embeddings for each node in the graph. In order to do so, we simply start with a one-hot vector representation of each node in a graph. Subsequently, for each node, a single-layered perceptron maps its vector into a dense vector, and then updates it iteratively as a function of itself and the representation of its neighboring nodes. More iterations allow the embedding of each node to capture more local information as per the graph structure. After a few such update iterations, the dense vectors of nodes belonging to the graph are max-pooled to compute the final embedding of the graph.

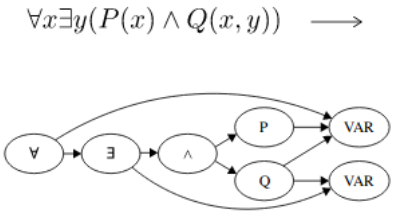

Figure 1. Example of a graph-based embedding, from [5].

1.2. Challenges and Contribution

Our main contribution lies in reproducing the results in [5]. The main challenges that we face are three fold -

1. We use the HolStep data set (see [4]), to carry out the experiments. This data set has its own syntax, and hence it requires parsing in order to create a graph out of a formula [4]. Building this parser was highly non-trivial as it required going through the HolStep documentation and experimenting with different interpretations of the syntax to arrive at the right parsing technique.

2. Authors in [5] do not provide all the hyperparameters that they used in training the neural networks. Hence, obtaining the same accuracy as the paper required a lot of effort in tuning the hyperparameters to their appropriate values. More details on this can be found in Section 3.2.

3. The training time for our neural network was very large with a single GPU (several days). Therefore, it was challenging for us to debug the code and try different hyperparameters. We were able to come really close to the test accuracy results of [5], but only in certain cases. The details of the differences between our results and the results in [5] are described in Section 4.

Note that, we have independently written all the code to reproduce the results in [5]. There is a repository available online to reproduce the results in [5][5]. This code trains faster than our implementation, but we do not borrow any code from this repository.

The rest of the report is organized as follows: Section 2 gives the algorithmic structure of the graph-based embedding technique. We summarize our experimental results in

[1] `ftp://mizar.uwb.edu.pl/pub/system/i386-linux/mizar-7.13.01_4.181.1147-i386-linux.tar`

[2] `http://cl-informatik.uibk.ac.at/cek/holstep/`

[3] We use the words/phrases mathematical statement, formula, statements, and mathematical formula interchangeably in this report.

[4] `https://www.cl.cam.ac.uk/~jrh13/hol-light/reference.pdf`

[5] `https://github.com/princeton-vl/FormulaNet`

Section 3. We also provide a short discussion on the differences between our results and the results in [5] in Section 4. Finally, we conclude the report with Section 5.

2. Method

In order to classify a conjecture-statement pair as relevant or irrelevant, it needs to be translated into a form that is suitable for processing by a neural network. In this section, we will discuss all of the pieces involved in turning the mathematical statements into a binary random variable. First, the formulas are parsed and represented as graphs. Then, we perform a neural-network based graph embedding to represent each graph as a vector. Finally, the vectors corresponding to the statement and conjecture are passed into a small binary classification neural network.

2.1. Creating a graph

The first step in our process is to take a formal mathematical statement and represent it as a graph. These statements are provided in the form of plain-text mathematical formulas, each of which is valid syntax for the HOL Light ATP system, in order to make parsing unambiguous. We broadly classify each term in a formula as a variable, variable function, constant, operator, or identifier. To better explain each of these types of terms, consider the following example:

$$\forall f\,(\exists x\,(f(c,x) \wedge P(x,f))), \tag{1}$$

Here, $\forall, \exists$ are quantifiers, which introduce variables and quantify how they are to be used in the formula. We treat these quantified variables as functions when they applied as functions in the formula, as with the first occurrence of f. Otherwise, we only classify them as variables. Therefore, both occurrences of x and the second occurrence of f are treated as variables. P and c are constants. Finally, $\wedge$ is an operator, an element of the base syntax of the ATP system.

Functions and operators combine one or more expressions into larger pieces of logic. These then are combined further in a recursive process, naturally creating a tree-like structure in each formula. If we were to simply treat an expression as a sequence of symbols, we might lose the information encoded by this structure. In the paper [5], the authors create a graph to capture this structure. We have used the same graph construction method.

The overall graph construction process is shown in Figure 2. Intuitively, we first parse the formula into its natural form as a tree. The data-set uses parentheses in a way that makes this step easy to perform; nearly each branch in the tree corresponds to one set of parentheses. This results in many duplicate nodes that logically refer to the same thing in the formula. In order to capture this, we merge all instances of the same variable into one node. Finally, variables are renamed to "VAR", and variable functions are renamed to "VARFUNC." After merging nodes corresponding to the same variable, the names of the variables become irrelevant to the semantic interpretation of the formula.

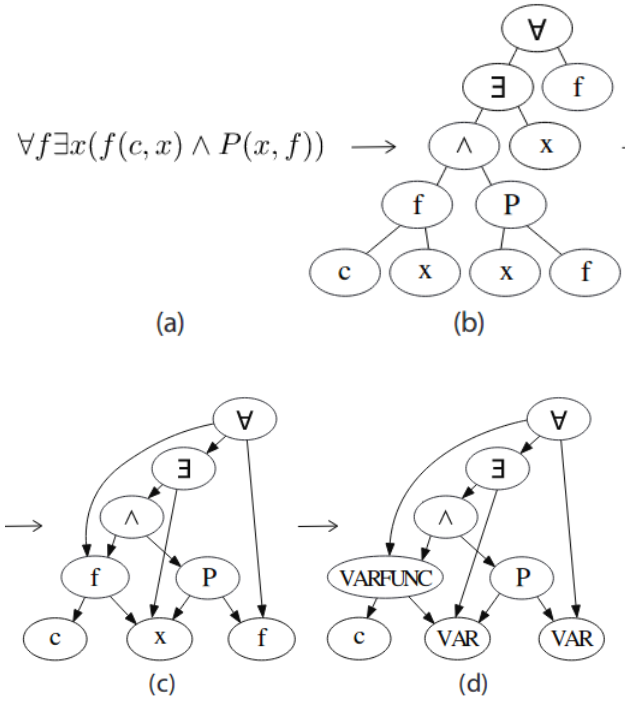

Figure 2. Figure showing different stages of conversion from the formula to Graph, from [5].

Given a formula s, a more detailed explanation of the recursive algorithm to construct a graph $G_s = (V_s, E_s)$ is as follows [5]:

1. If the statement is a single symbol (like constant value or variable value) α, there is one node in the graph given by $V_s = \{\alpha\}, E_s\, \emptyset$.

2. If the statement is a variable function of n different statements $s_1, \ldots, s_n$ (i.e. $s = f(s_1, \ldots, s_n)$, we create a new node f, and connect that by an edge to the head node of graphs $G_{s_1}, \ldots, G_{s_n}$. Then we merge the leaf nodes that are the same token (like constant value) in the graph. Therefore we return the graph $G = (V, E)$

$$V = \bigcup_{i=1}^{n} V_{s_i} \cup \{f\},\ E = \bigcup_{i=1}^{n} E_{s_i} \cup \bigcup_{i=1}^{n} (f, \text{head}(G_{s_i})). \tag{2}$$

   This is followed by $MERGE_C(G)$.

3. If the statement is a quantifier ϕ applied to variable x followed by any formula t, i.e. $\phi_x t$, we add nodes ϕ and x. Make x and the head node of t as the children to ϕ. Then we merge node x with all of its occurrences

in the graph of t. Finally, we rename the variable x to the token VAR.

Therefore, we create $G_s = (V, E)$ where

$$V = V_t, \{x, t\}, E = E_t, (\phi, \text{head}(G_t)), \bigcup_{v \in V_t[x]}^{n} (\phi, v)$$

$V_t[x]$ is the set of nodes in the graph of t which represent t. Output $RENAME_x(MERGE_x(G))$.

2.2. Embedding the Graph

In the previous section we described how one can create a graph which preserves the structure in a formula. The next step is to create an embedding from this graph, which serves as an input vector to the classification model.

In order to create the embedding, we initially associate each node in the graph with an one-hot vector. We have nodes of 1909 types (one each for VAR and VARFUNC, one for each operator, and one for each unique constant in the data-set, plus an extra called UNKNOWN for constants not present in the data-set), so the one-hot vector has dimension 1909. We then update the vector embedding of each node according to embedding of its neighbouring nodes.

As an example, consider the formula and graph from Figure 1. Note that although x and y have different nodes in the formula $\forall x \exists y(P(x) \wedge Q(x, y))$, they are of the same type, and therefore are initialized with the same one-hot vector. Nodes are updated as a function of their neighbors' vectors. As an example, the iterative update of node Q at step $t + 1$ is a function of the embeddings of node $`VAR_1, VAR_2$ at time t. The update is given by,

$$\begin{aligned} Q^{t+1} =& F_P^t(Q^t + \frac{1}{3}[F_I(\wedge^t, Q^t) + F_O(Q^t, VAR_1^t) \\ &+ F_O(Q^t, VAR_2^t)]) \end{aligned} \tag{3}$$

where, functions F_P, F_I, F_O are all represented by neural networks and they are learned concurrently with the classification model during training. Essentially, Q^t is added to the average of the functions of incoming nodes and outgoing nodes and then passed through F_P. Given a graph $G = (V, E)$ and node embedding x_v of node v, a general update rule would be,

$$\begin{aligned} \text{update-1:} \quad x_v^{t+1} =& F_P^t(x_v^t + \frac{1}{d_v}[\sum_{(u,v) \in E} F_I(x_u^t, x_v^t) \\ &+ \sum_{(v,u) \in E} F_O(x_v^t, x_u^t)]) \end{aligned} \tag{4}$$

where d_v is the sum of in-degree and out-degree of node v.

This algorithm can be run with any number of update steps. More steps may better capture the global structure of the graph, but makes training slower. After the desired number of update steps are completed, the final graph embedding is obtained by max-pooling all nodes in the graph.

After having a graph embedding, we concatenate the graph embeddings of the conjecture and statement and pass the result to the binary classification model. All the weights of F_P, F_I, F_O are trained concurrently with the classifier.

2.3. An order preserving embedding

The graph embedding described above is invariant to variable naming, but it has its shortcomings. It does not preserve the order in which the inputs are fed into the function. For example, suppose we have a function $f(x_1, x_2, x_3)$ with a node f with edges connecting to nodes x_1, x_2, x_3. Then the function F_I, F_O is additive on the edges $f - x_1$, $f - x_2$ and $f - x_3$. In this subsection, we define an embedding which uses the ordering of the inputs to create a feature.

First, we need to define something called a 'treelet' of a graph. Let us understand this through an example we introduced in the previous paragraph. Let us order the inputs according to their rank which defined as 1 for x_1, 2 for x_2 and 3 for x_3. Then the 'treelets' for node f is the set of triplets $\{|\{x_1, f, x_2\}, \{x_1, f, x_3\}, \{x_2, f, x_3\}\}$. More Formally, if the vertex set is V, then the treelets is the set with elements in $V \times V \times V$ such that for each element (x, y, z), y is the parent of x and z and rank of x is less than the rank of z.

We update the node embedding of node v in the following way.

$$\begin{aligned} v^{t+1} =& F_P^t[v^t + \frac{1}{d_v}[\sum_{\text{incoming edge } e} F_I(e) + \sum_{\text{outgoing edge } e} F_O(e)] \\ &+ \frac{1}{e_v}[\sum_{treelet \text{ with v as left child}} F_L(treelet) \\ &+ \sum_{treelet \text{ with v as parent}} F_H(treelet) \\ &+ \sum_{treelet \text{ with v as right child}} F_R(treelet) \ ] \ ] \end{aligned} \tag{5}$$

where d_v is the degree of the node v and e_v is the number of treelets in which node v appears.

2.4. Neural Network architecture

We used a total of 6 neural networks for the classification task.

All neural networks can be described in terms of a block consisting of a fully connected layer followed by Batch Normalization and a ReLU activation function as in Figure 3. Let us call this block B.

Let us understand the whole neural network using an example. Suppose we are training the optimization algorithm with a batch size of 10. So we input 10 statement

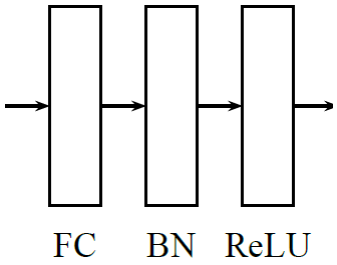

Figure 3. The basic neural network block.

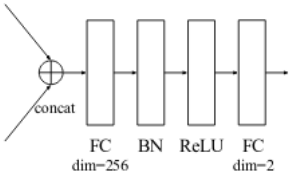

Figure 4. The binary classification neural network, from [5].

graphs and 10 conjecture graphs at one time to the the neural network. Suppose each graph has 60 nodes each. Each node has a corresponding one-hot vector of 1909 dimension. Thus for each batch of 10 statements, we pass a vector of size 600×1909 through a fully connector layer to get an output of dimension 600×256. We remove the empirical bias and variance from these 600 vectors in the batch normalization process. Note that here, we are doing both inter-graph and intra-graph batching. The output of the size 600×256 is fed through a ReLU network with output size 600×256. The first 60 rows correspond to the 1st graph, $60 - 120$ correspond to the second, and so on. Now, if we have zero order update, then for each of the 10 graphs we simply take a coordinate-wise maximum among all 60 nodes(max-pool). A similar process is followed for the 10 conjecture graphs (i.e. it is passed through a block type B) and we have a 10×256 conjecture embedding. Then we concatenate the statement conjecture pairs to form a 10×512 (conjecture,statement) pair embedding, which is fed into the classification neural network (shown in Figure 4). The classification network has a block B followed by a fully connected layer with output dimension 2 (corresponding to two classes). In the zero-order update, we do not use the structure of the graph.

If we have a $1-$order update rule from equation (4), we proceed as follows. After the output of size 600×256 of 10 statements from the ReLU network, we update the node embedding of each node according to the graph structure as in (4). The structure of F_P consists of one block B and both F_I and F_O have two blocks B. This is shown in Figure 5. Note that in one update step the function F_P, F_I, F_O remain the same for all nodes in a particular graph. The same

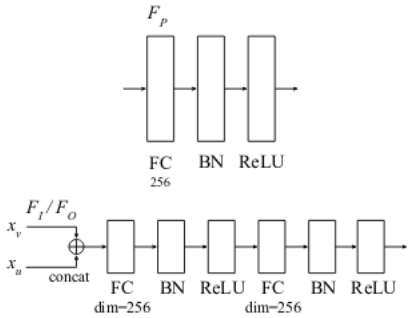

Figure 5. Neural network architecture for functions F_P, F_I, F_O, from [5].

process of update is followed for the 10 conjecture graphs. It is then max-pooled to generate a matrix of size 10×256 which is fed to the classification model, concatenated with the conjecture embedding. With more than one update step, we cascade multiple F_P, F_I, F_O to create an embedding.

For the order preserving functions F_L, F_H, F_R we concatenate the three inputs and then pass it through two cascaded blocks of B as shown in Figure 6 below.

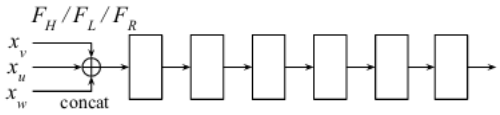

Figure 6. Order preserving update functions F_L, F_H, F_R, from [5].

3. Experimental Results

3.1. Dataset

We trained and tested our model on the HolStep dataset [4] which consists of about 2 million training examples and 196 thousand testing examples where all formulas are in higher-order logic. Each example is a (conjecture, statement, label) triplet where the label indicates whether the statement is useful (1) or not useful (0) in proving the conjecture. Half of the statements are useful in proving the conjectures, so an algorithm guessing a single label would obtain 50% accuracy. We partitioned the training examples so that 7% is left out for validation.

In the training dataset, there are 1908 unique tokens that appear in more than two formulas. Any token that appears less than twice or never appears in the training dataset is mapped to an "UNKNOWN" token. Each token is represented by a 1909-dimensional one-hot vector which is mapped into a 256-dimensional dense vector using a fully connected layer followed by batch-normalization and a ReLU activation function as in fig 3. We note that this is our interpretation of the the one-hot to dense mapping

function, and may not be how the original authors in [5] implemented the one-hot to dense map, as this layer is not clearly stated.

3.2. Model Training

There are two aspects of batching by the nature of the problem setup: for each graph there are multiple node pair/triplets that serve as inputs to the network as seen by (4) and (5), and we may also feed in multiple (conjecture, statement) pairs into the network to obtain a more accurate gradient estimate and batch-normalization estimate. We refer to these as intra-graph batching and inter-graph batching respectively. Intra-graph batching depends on the number of nodes in each graph and we are unable to control this batch-size. Thus, from here on, we refer to batch-size as the inter-graph batch size, or the number of distinct (conjecture, statement) pairs we feed into the network .

We used the same hyperparameters as in [5] except those that were unspecified. We observed that the networks perform significantly better when trained over an inter-graph batch size of ≥ 16 as opposed to smaller batch sizes (e.g. $2, 4$, and 8). This is expected since it is well known that small batch sizes are detrimental when using Batch Normalization. For comparison, when we trained the 0^{th} order model with a batch size of 4 the test error was approximately 42% at the end of 5 epochs, whereas the model trained with a batch size of 16 and 32 both achieved a test-error of 31%. Larger batch-sizes improved the test-error monotonically. Following this observation, we trained the 1^{st} order model with a batch size of 32.

Since the 0^{th} order model consists of two trainable networks, namely one-hot to dense mapping and the classification network, both forward operation and backpropogation were significantly faster than any higher-order models. Training the 0^{th} order model over 5-epochs took only half a day, whereas we estimate training a 1^{st} order model to take around 15-20 days, all on a Titan X GPU.

While we implemented FormulaNet which uses the update rule (5), training FormulaNet takes approximately twice as long as training FormulaNet-basic (4). We were unable to train FormulaNet to the extent of obtaining meaningful results, so we report only the results of FormulaNet-basic.

3.3. Main Results

Our 0^{th} order model was able to achieve 28% validation and 30.8% testing error when trained with a batch-size of 32, and 25.65% validation error with 27.85% test error using a batch size of 256. The first order model did not complete training but obtained 18.27% test error by the end of $1/5$ epoch (12,000 batches). These results are summarized in Table 1.

Number of Steps	Batch Size	Epochs Trained	Test Error (%)
0	16	5	30.8
0	32	5	30.4
0	256	5	27.85
1	32	1/5	18.27

Table 1. Results.

4. Discussion

Our test error for the 0^{th} order model performed worse than stated in [5] by a margin of 9%. Had we trained the 1^{st} order model with a larger batch-size, it seems we would have obtained similar results as obtained in [5]. Possible explanations for why we attained worse results may be different hyperparameters (batch size) or slightly different pre-processing. The batch-size used was not specified by the original authors, and our pre-processing resulted in more than 1908 distinct tokens which is the number of tokens the authors argue were in the training dataset. This is possible because datapoints in the training dataset are randomly selected to be partitioned for the validation dataset.

5. Conclusion / Future Work

Structured prediction is a tool used to improve the accuracy when outputs are correlated. Because conjectures often cannot be proved using single statements but rather are proved with a collection of statements, structured prediction seems to be a plausible alternative to improve the accuracy of premise selection because most conjectures can be proved using different statement collections. Take for example the conjecture that $\sqrt{2}$ is irrational. The definition of rational numbers can be used to prove this conjecture by contradiction. Alternatively, this conjecture can be proved by using the following statements [3]:

1. s_3 : If p is prime, $\sqrt{p}$ is a root of $f(x) = x^2 - p$.
2. s_4 : Gauss's lemma:

$$-f(x) = x^2 - p$$

has no rational roots.

Denote the four formulas $s_1 =$ 'a rational number can be written in the form of p/q where $p, q \neq 0$ are integers', $s_2 =$ 'definition of contradiction', and the two statements s_3, s_4 respectively which form the alternative proof to $\sqrt{2}$ is irrational mentioned above. It is obvious that there is some *structure* inherent in proving a conjecture, namely that $(s_1, s_2), (s_3, s_4)$ are useful in proving the conjecture, but all other pairs are not. By considering the problem of finding statements relevant to proving a conjecture as a normal binary classification task is wasting the correlation between statements in proving a conjecture.

To further motivate the use of *structured inference* for premise selection, we note that [5] considered an *unconditional* setting in which the classification network only takes in a statement as opposed to a (conjecture, statement) pair to determine whether the statement is useful. The unconditional and conditional classification accuracy were surprisingly very similar for both FormulaNet-basic and FormulaNet: 89% for FormulaNet-basic and 90%, 90.3% respectively for FormulaNet. This suggests that certain statements are just highly relevant in proving theorems and that training over (conjecture, statement) pairs does not contain the necessary structure of proofs, which most likely consists of multiple statements per conjecture. It would be interesting to see how structured inference can improve the accuracy of identifying which statements are relevant in proving a conjecture.

Another extension could be the use of different architectures such as those which taken in graph-inputs [2]. This would offer the advantage of directly dealing with graphs for neural network inputs rather than using distinct functions for each pair as in (4) and (5). This approach is not only simpler than the update rule used in (4) and (5) but also may speed up the training due to fewer hidden layers.

Acknowledgement

We would like to thank our friend and labmate Joseph Lubars for helping out with the project, especially the help he extended in building the parser for the HolStep dataset.

References

[1] A. A. Alemi, F. Chollet, G. Irving, C. Szegedy, and J. Urban. Deepmath - deep sequence models for premise selection. *CoRR*, abs/1606.04442, 2016.

[2] P. W. Battaglia, J. B. Hamrick, V. Bapst, A. Sanchez-Gonzalez, V. F. Zambaldi, M. Malinowski, A. Tacchetti, D. Raposo, A. Santoro, R. Faulkner, Ç. Gülçehre, F. Song, A. J. Ballard, J. Gilmer, G. E. Dahl, A. Vaswani, K. Allen, C. Nash, V. Langston, C. Dyer, N. Heess, D. Wierstra, P. Kohli, M. Botvinick, O. Vinyals, Y. Li, and R. Pascanu. Relational inductive biases, deep learning, and graph networks. *CoRR*, abs/1806.01261, 2018.

[3] F. M. (https://math.stackexchange.com/users/4284/fredrik meyer). Irrationality proofs not by contradiction. Mathematics Stack Exchange. URL:https://math.stackexchange.com/q/20567 (version: 2014-11-11).

[4] C. Kaliszyk, F. Chollet, and C. Szegedy. Holstep: A machine learning dataset for higher-order logic theorem proving. *CoRR*, abs/1703.00426, 2017.

[5] M. Wang, Y. Tang, J. Wang, and J. Deng. Premise selection for theorem proving by deep graph embedding. *CoRR*, abs/1709.09994, 2017.