Automated Theorem Proving with Graph Neural Networks

Daniel Jenson, Julian Cooper and Daniel Huang

March 20, 2023

Abstract

By leveraging recent advances in Graph Neural Networks (GNNs), we were able to exceed the performance of TreeLSTM-based models in the CoqGym dataset. The CoqGym dataset consists of roughly 189 thousand mathematical and software verification proofs that use the Coq automated (ATP) and interactive (ITP) theorem prover. Our best model proves 19.4% of proofs in the ZFC dataset, which consists of 237 proofs regarding Zermelo–Fraenkel set theory. By comparison, ASTactic, the CoqGym author's model proves 14.8%. Our model uses a combination of a GNN deep encoder with a modified version of a tactic decoder to generate valid tactics for solving proofs.

1 Problem Statement

Theorem proving is a difficult, lengthy process, which has historically been the sole employ of mathematicians. Recent engineering and computer science advances have seen the advent of theorem proving assistants. These are software programs that aid mathematicians in proving statements by providing a formal syntax for composing proofs and a verification process for validating them.

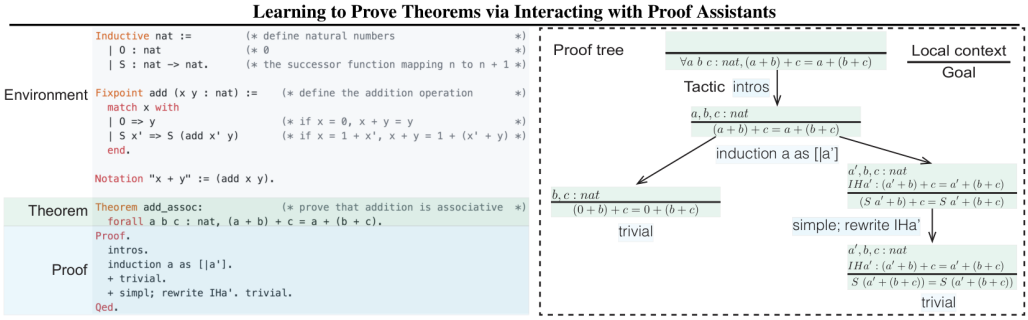

Figure 1: Automated Proof Assistant: an example of the induction tactic [7]

More recently, some of these software programs have begun to provide automated theorem proving (ATP) components, which are capable of automatically selecting tactics and proving simpler subgoals. Since 2019, these ATP modules have been further augmented with neural networks, creating a new class of ATPs called Neural Theorem Provers (NTPs). NTPs are often structured as an encoder-decoder, which consists of an encoder that embeds goals in a latent space and a decoder that uses these representations to produce a distribution over valid tactics or proof techniques. In this project, we extend one of these NTP frameworks.

In this framework we start with a goal (statement to prove) and our agent analyzes existing data (solved proof steps) to develop a model that predicts the next best action, which is the application of a proof tactic, e.g. mathematical induction. This action is then submitted to the environment, which applies the tactic to our goal and returns `SUCCESS` if solved, or `UNSOLVED` along with any subgoals that remain to be solved. This interaction progresses the proof one step. The resulting subgoals, along with the context (relevant axioms and proven premises), form our new state. This information is fed into the encoder, which produces new embeddings that the tactic decoder uses to generate the next tactic. This repeats until all subgoals are empty and the proof is complete, or the process terminates because a maximum number of tactics have been applied or it has reached a maximum time limit.

Since our agent's understanding of the problematically best tactic to submit at each step is imperfect, there is model uncertainty. Moreover, the state space is exponentially large with a branching factor much larger than the game of Go [2]. The state space consists of MDPs at two levels. At the outer level, the state space consists of proof trees, where each action applies a tactic to create subgoals and generate a new proof tree. At the inner level, the state space consists of tactic trees, where each action adds a node to the tactic tree until it represents a fully-formed tactic that can be applied. The outer level is explored with depth-first search, while the latter uses beam search in conjunction with an attention mechanism.

2 Algorithm & Methodology

The NTP for our project is composed of two parts: (1) a graph neural network (GNN) encoder and (2) the reinforcement learning agent, which includes a tactic decoder (detailed below). The encoder and decoder are trained with teacher forcing using human proofs. At test time, the reinforcement learning agent uses the output of the model and applies estimated high value tactics to the current goal. It limits the number of tactic applications to 300 and the run-time limit to 10 minutes and is rewarded only when the proof goal and all of its subgoals are proved.

There are two principle baselines from existing literature: CoqGym [7] and HOList [5] CoqGym is based on the Coq theorem proving language and HOList is based on the HOL Light theorem prover. Both implement similar environments for generating states given submitted tactics, but they differ substantively in (a) their representation of goal and premise statements and (b) their encoder-decoder models. Both use abstract syntax trees (ASTs) as their root data type, but Coq uses higher level expressions in the AST. On the other hand, HOList uses a lower-level representation but modifies these ASTs using subexpression sharing, leaf sharing, and random edges in their graph representations.

The original Coq NTP paper uses TreeLSTMs to represent the hidden state for terms, while HOList's best model uses a 12-hop GNN. We used CoqGym instead of HOList because the ASTs represented at each proof step capture higher level logic than HOList without requiring significant modifications. Furthermore, until recently, the HOList dataset was not available, and a large portion of the original source code remains unreleased.

Another key difference between these models is the scope of tactic generation. HOList limits the number of applicable tactics to 41, while CoqGym's ASTactic can generate a practically infinite number of tactics. In doing so, the HOList GNN can learn to predict which tactics are best where those 41 tactics are viable. However, it inherently limits the flexibility of tactic application. On the other hand, CoqGym's ASTactic can generate any number of tactics. However, these tactics are constructed iteratively, expanding a partial tree until it becomes a valid tactic. This process is slow and expensive. The model also has to learn a much more complicated mapping from current (proof) state to valid tactic.

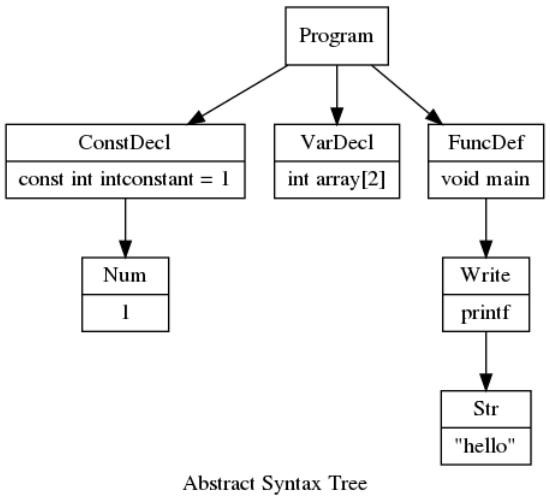

Figure 2: Coq's Abstract Syntax Tree (AST) [7]

2.1 Extensions to the encoder and agent

The following details algorithm modifications we made to the term encoder and reinforcement learning agent.

1. **Graph neural network deep encoder**: The CoqGym paper implemented a TreeLSTM term encoder. We want to improve upon this with a more expressive Graph Neural Network, taking inspiration from the encoder design proposed in Google's HOList paper. Our implementation used two graph convolution layers (2-hop) with ReLU activation and dropout. We experiment with two design choices: (a) GraphSage [4] vs. Graph Attention (GAT) [6], and (b) global mean vs. max pooling. GraphSage and GAT differ in how they pass and aggregate messages in the convolution layer. GraphSage assumes messages from neighboring nodes should be equally weighted in importance, whereas GAT allows these weights to vary using an attention mechanism. Global mean and max pooling are two different methods for taking node embeddings we get from convolution layers and generating a single graph embedding. For our problem, the node embeddings each represent individual nodes of a given term AST (single component of the representation of a premise statement, e.g. $\in$ operator), and so pooling across these gives us an embedding for the entire term AST (premise statement, e.g. $a, b, c \in \mathbb{R}$). Mean and max pooling offer different types of information. Mean pool yields an equal weighting on all the nodes, and the embeddings will be expressing what a representative "average node" would look like for the AST while max pooling would highlight individually expressive components of the embedding features over all the nodes. Note, we did not consider global sum pooling since our ASTs vary wildly in size and so some form of normalization would have been required. Differential Pooling is another mechanism we looked into and believe to be very promising, but had to leave for future research.

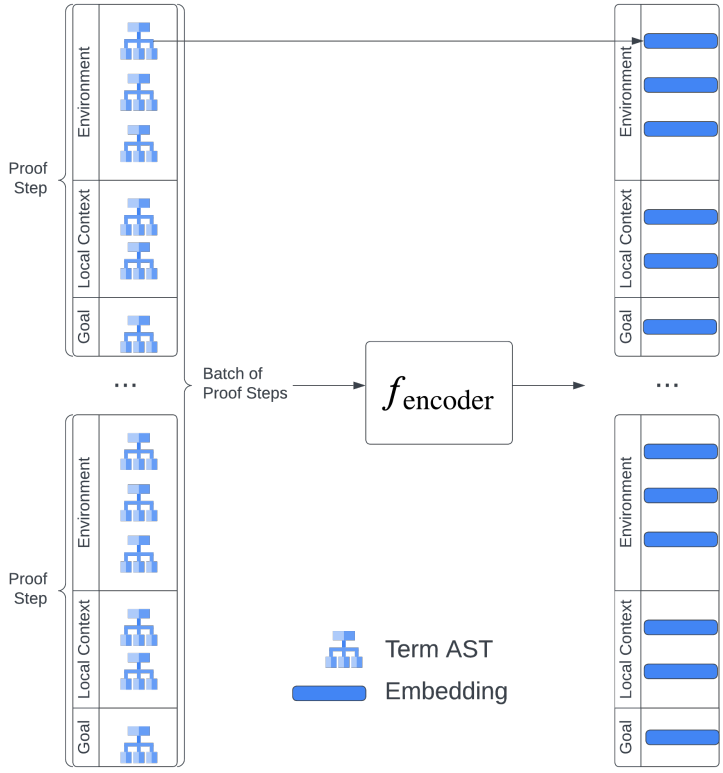

Figure 3: GNN deep encoder input and output

2. **Rich attention for the tactic decoder**: For the tactic decoder, CoqGym used a gated recurrent unit (GRU) with a simple attention mechanism to weight the contribution local context and environment information. The original attention mechanism was a simple two layer feed forward network with a ReLU activation between them. Based on guidance from [9] and [1], we made three changes: (1) we replaced the ReLU activation with a PReLU, (2) we increased the number of layers from two to three, and (3) added layer normalization. In

our experiments, this richer attention mechanism had a significant effect on final performance. A PReLU is similar to a ReLU but introduces a learned parameter to allow activation values below zero [9]. Increasing the number of layers allows the network to learn more complex functions, and layer normalization largely prevents gradient vanishing and exploding [1]

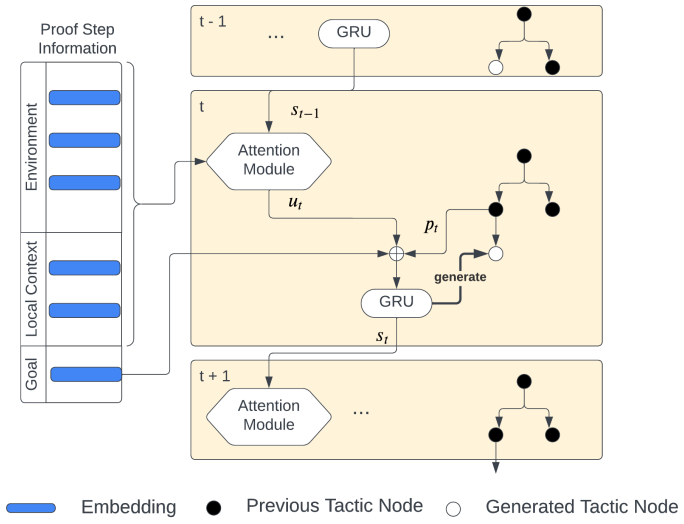

Figure 4: GNN deep encoder input and output

3. **Shallow embeddings for node (term) types**: There are multiple ways to initialize node embeddings for our graph neural network encoder: fixed integer, one-hot encoding, integer embedding, centrality-based approaches (e.g. degree, clique reference), or random-walk-based approaches (e.g. DeepWalk, HOPE) [3]. We decided to make use of GraphGym's integer feature encoder [8] which picks semi-orthogonal directions in n-dimensional space for each of our $m > n$ input nodes. As we train the model, these feature vectors are shifted such that similar nodes become closer in the embedding space. This allows our model to transfer learning between embeddings in a localized region of the embedding space and improves the model's capacity to generalize over these features. With sufficient training data, this decreases the memory footprint relative to one-hot encoding, and improves the representation of features in a more compact subspace.

Size of our training set: Deep neural networks are very data hungry and typically require very large datasets to perform well. CoqGym extracts roughly 189 thousand proof steps to in a teacher-forcing fashion when constructing tactics. Extracting the baseline proof steps using the original ASTactic proof steps requires roughly four CPU hours multiprocessed across 12 cores. However, our GNN model requires a richer representation in order to generate embeddings. Accordingly, we had to extract node features as well as edge indices with the proof steps so that we could construct a graph for every term in every proof step. This increased the data extraction time from approximately 8 hours to nearly 2.5 days. This figure also reflects a number of optimizations (detailed below) that we had to implement. Our original basic extension required nearly 5 days to extract all the data in a format PyGeometric could use. There are some proofs and proof projects that are significant outliers, which stalls extraction for a very long time as all intermediate graphs are constructed and serialized. Given this, we trained models on various size training datasets to understand the effect of the dataset size on performance in the test set.

2.2 Extensions to the Coqgym software pipeline

While our principle focus was on encoder, decoder, and agent extensions, we realized early on that a significant amount of development would be dedicated to improving the development cycle and overcoming inefficiencies in CoqGym's base implementation. In addition to the aforementioned modeling improvements, we also made the following improvements to the underlying platform:

1. **Proof step data augmentation and batching**: PyGeometric graphs require node features and edge indices for construction. Accordingly, we had to extract that data and serialize it in a

format that PyGeometric could use. This dramatically increased the size of the serialized proof steps. Furthermore, this required that we modify the dataloader so that it could use an object that carried elements we required to encode term ASTs as graph embeddings as well as all the additional information the tactic decoder required to utilize this embedding in building tactics. A particularly tricky part of this implementation involved retooling the batching process. Each proof step might contain hundreds of environment and local context terms, as well as a goal term. Each of these terms is represented as an tree graph (AST) in PyGeometric. So, a batch, which consists of several proof steps, often contains thousands of graphs. Ensuring that the environment, local context, and goal embeddings are properly extracted and attributed to the correct proof step was a subtle enterprise, especially since we needed to combine these encoded terms with the metadata that was part of the original batch.

2. **Parallelized proof extraction and evaluation**: The scripts provided for proof extraction and evaluation processes each proof in series. This causes the extraction process on the entire dataset to take approximately 12 hours (over 5 days for extracting all PyGeometric graph object information) and much longer for evaluation (on the order of days or weeks). These processes are trivially parallelizable, so we implemented a multiprocessing script to process each proof in parallel. With further optimizations to their iteration process, the data extraction ended up taking roughly 2.5 days including all PyGeometric graph objects. However for evaluation, the optimizations did not bring down the evaluation time as the process is significantly limited by the serial process of searching the proof tree. This is why we limit our testing for this project to the subset of proofs in the ZFC project. At 237 proofs, we felt the ZFC project represented a sufficiently large dataset to compare the performance of models employing different techniques and inform the evolution of our experiemnts.

3 Results & Discussion

For evaluating the effectiveness our various model extensions, we selected an example proof library (known as ZFC) with 237 human-written proofs as our baseline. These proofs are taken from Zermelo–Fraenkel set theory which is a well-studied branch of mathematics. In future research we would like to expand this test set to include all proof libraries used to generate CoqGym's published benchmarks, but due to limited time and computational resources, we narrowed the scope of our testing pipeline for this project.

We run experiments testing a variety of extensions on 65% of the training dataset, including: (a) GraphSage vs. GAT term encoder, (b) global mean vs. max pooling, (c) one-hot vs. integer feature encoding, and (d) rich vs. simple attention in the decoder.

Ref	Conv. Type	Pooling	Attention	Embedding	Accuracy (%)	Correct (/237)
1	GraphSage	max	rich	integer	**17.7%**	42
2	GraphSage	mean	rich	integer	13.1%	31
3	GAT	max	rich	integer	11.4%	27
4	GAT	mean	rich	integer	**15.2%**	36
5	GAT	mean	rich	one-hot	13.1%	31
6	GAT	mean	simple	integer	10.6%	25

Table 1: Accuracy results for encoder-decoder model variants using ZFC proof library.

From our experimental results on the partial dataset, we learned a lot. First, by comparing models 4, 5 and 6, we find that both our rich attention and integer feature encoder modifications were valuable improvements. Second, it is not clear whether global mean or max pooling is more appropriate for this problem since they perform differently for each of the different convolution layer types. And third, while GraphSage appeared to perform better than GAT, then delta was small enough that both appeared in our two best models.

For each of the results in the Table 1, we show the results from our best performing epoch (up to 10). CoqGym used 5 epochs for its ASTactic model and they commented that their performance did not improve meaningfully after epochs 3-4. While we also observed a similar trend, we did find substantial improvements in epochs 5-10 for some of our models, likely because the GNN-based models had far more trainable parameters (see figure 5).

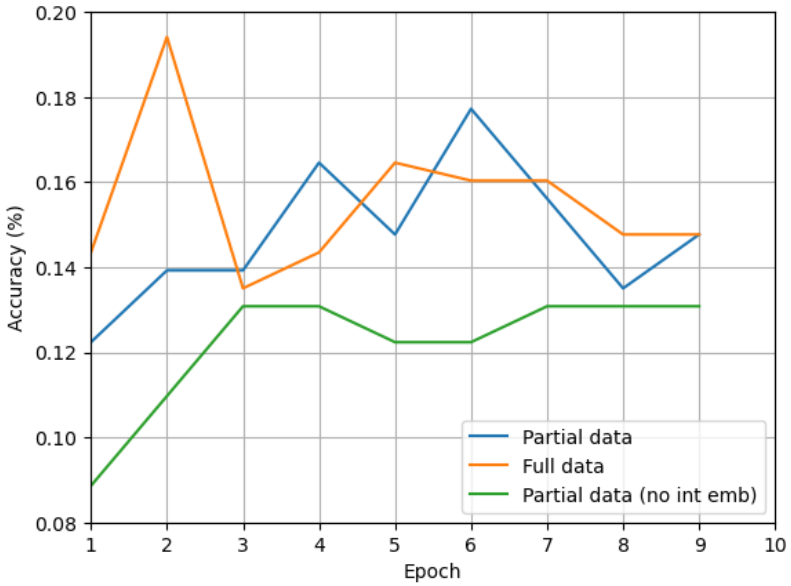

Figure 5: Performance for three example models over epochs 1-9

Having experimented with different modifications, we picked the two best performing models (highlighted in Table 1) to train on the full dataset and benchmark against CoqGym's ASTactic results for the ZFC proof library. We present these results in Table 2.

Ref	Conv. Type	Pooling	Attention	Embedding	Accuracy (%)	Correct (/237)
ASTactic	n/a	n/a	n/a	n/a	14.8%	35
Model #1	GraphSage	max	rich	integer	**19.4%**	46
Model #4	GAT	mean	rich	integer	17.3%	41

Table 2: Benchmark accuracy results for our two best performing models trained on the full dataset

4 Conclusion

In this project, we have found that GNNs are a viable and performant alternative to TreeLSTMs in generating proof tactics and proofs. Our best model, a max-pooled, integer-feature embedded, rich attention GraphSage neural network achieved 19.4% on the ZFC test set, while ASTactic achieved 14.8%. While this is a modest improvement, we view this as a promising start, as we have a number of enhancements we wish to test that we believe will further increase the expressive power of the generated embeddings.

5 Next Steps

Given that nearly half of our development time was devoted to software enhancements rather than model experimentation and testing, we plan to continue working on this next quarter. In particular, we plan on experimenting with ID-GNNs, which are provably the most expressive GNNs, as well as Differential Pooling, which creates a more intelligent graph embeddings from node embeddings using hierarchical clustering. We also plan on using more "hops," in our network, as HOList saw improvements up to 12 hops and we only used 2. Lastly, we believe we can improve the agent further by experimenting with different tactic search mechanisms, perhaps using Monte Carlo Tree Search in lieu of depth-first search. We also plan on experimenting with another theorem proving program, Isabelle, and it's publicly available Archive of Formal Proofs (AFP), which is notably easier to manipulate than CoqGym's proofs and serialized internals.

Contributions: Dan Jenson owned setup of our remote server, multiprocessing and software extensions to the CoqGym pipeline. Julian Cooper owned development of extensions to the term encoder. Daniel Huang owned development of updates to the term decoder and assisted in optimizing data augmentation and extraction. Everyone contributed to brainstorming GNN and RL design choices, testing and evaluation methodology and sense checking results.

References

[1] Jimmy Lei Ba, Jamie Ryan Kiros, and Geoffrey E. Hinton. *Layer Normalization*. 2016. arXiv: 1607.06450 [stat.ML].

[2] Yutian Chen et al. "Bayesian Optimization in AlphaGo". In: *CoRR* abs/1812.06855 (2018). arXiv: 1812.06855. URL: http://arxiv.org/abs/1812.06855.

[3] Chi Thang Duong et al. "On Node Features for Graph Neural Networks". In: *CoRR* abs/1911.08795 (2019). arXiv: 1911.08795. URL: http://arxiv.org/abs/1911.08795.

[4] William L. Hamilton, Rex Ying, and Jure Leskovec. "Inductive Representation Learning on Large Graphs". In: *CoRR* abs/1706.02216 (2017). arXiv: 1706.02216. URL: http://arxiv.org/abs/1706.02216.

[5] Aditya Paliwal et al. *Graph Representations for Higher-Order Logic and Theorem Proving*. Sept. 12, 2019. DOI: 10.48550/arXiv.1905.10006. arXiv: 1905.10006[cs, stat]. URL: http://arxiv.org/abs/1905.10006 (visited on 01/20/2023).

[6] Petar Veličković et al. *Graph Attention Networks*. 2018. arXiv: 1710.10903 [stat.ML].

[7] Kaiyu Yang and Jia Deng. *Learning to Prove Theorems via Interacting with Proof Assistants*. May 21, 2019. DOI: 10.48550/arXiv.1905.09381. arXiv: 1905.09381[cs,stat]. URL: http://arxiv.org/abs/1905.09381 (visited on 01/20/2023).

[8] Jiaxuan You, Rex Ying, and Jure Leskovec. "Design Space for Graph Neural Networks". In: *NeurIPS*. 2020.

[9] Jiaxuan You, Rex Ying, and Jure Leskovec. "Design Space for Graph Neural Networks". In: *CoRR* abs/2011.08843 (2020). arXiv: 2011.08843. URL: https://arxiv.org/abs/2011.08843.