Deep Learning Recommendations for the Theorem Prover ACL2

Kyle Thompson[1][0000−0002−2868−7612], Ruben Gamboa[2], Matt Kaufmann[3], Eric McCarthy[4], Eric Smith[4], and Dennis Sun[1]

[1] California Polytechnic State University, San Luis Obispo CA 93405, USA
rkthomps@calpoly.edu dsun09@calpoly.edu
[2] University of Wyoming, Laramie WY 82071, USA ruben@uwyo.edu
[3] University of Texas, Austin TX 78712, USA (retired) kaufmann@cs.utexas.edu
[4] Kestrel Institute, Palo Alto CA 93405, USA mccarthy@kestrel.edu
eric.smith@kestrel.edu

Abstract. ACL2 is an industrial-strength interactive theorem prover. Although ACL2 can automatically prove simple theorems, deep theorems can require substantial expertise and time from users. We have developed a machine learning system that automatically provides recommendations to help users complete proofs. We trained the machine learning models by systematically breaking proofs in the public libraries of ACL2 and learning to predict the fixes. The models are based on the transformer architecture from deep learning that has found success in natural language processing applications. On a held-out test set, our system predicts the fix 18% of the time, and the fix is among the top 10 recommendations 38% of the time. Such a system can be integrated into a tool that automatically tries the top-K recommendations.

Keywords: ACL2 · Proof Repair · Deep Learning · Formal Methods.

1 Introduction

ACL2 proof advice backed by machine learning can potentially benefit all ACL2 users. Specifically, it may reduce the time needed to find proofs, even for experts. And it may be especially helpful for new users, either by reducing the amount of expertise needed to prove a given theorem, or by directing them to helpful rules and libraries. Such advice may be helpful for both creating new proofs and repairing existing ones. We have built a proof advice tool backed by deep learning models that recommends actions a user can take to prove a failed ACL2 conjecture. The tool can be accessed via a web interface, where users may copy and paste their failed proof attempts. We have also built a utility that runs in the ACL2 session and automatically tries recommendations from the models.

In a failed proof attempt, ACL2 gives the user important information about the proof attempt including subgoals that ACL2 could not prove [7]. These subgoals are called checkpoints. Our models supplement the users' examination of failed proof attempts by giving actionable recommendations based on the reported checkpoints.

To obtain checkpoints for training, we systematically and automatically break proofs in ACL2's community libraries. We treat the problem of producing recommendations based on a broken theorem's checkpoints as a classification problem. We define the set of actions that our models can recommend to the user as a set of {action-type, action-object} pairs where the action-type indicates how to use the action-object. For example, take the recommended action (add-library, "arithmetic/factorial"). The "add-library" action-type indicates that the user should try including the indicated library in their ACL2 session, and the "arithmetic/factorial" action-object indicates which library to include. Each action-type roughly corresponds to a mechanism we have for breaking proofs in the public libraries. In general, they all remove different kinds of objects (hints, assumptions, libraries, etc.) from theorems or their proofs, to attempt to change them enough so that ACL2 can no longer prove them automatically. If this is the case, we know that undoing the removal (adding the hint or assumption, including the library, etc.) will allow ACL2 to complete the proof of the theorem.

2 Related Work

To advance the field of formal methods, there have been many efforts to reduce the number of interactions needed between a user and an interactive theorem prover (ITP) [8]. Parallel to ITPs are automated theorem provers (ATPs). ATPs have yielded impressive results for decades. Notably, the Robbins conjecture was solved in 1997 using an ATP called EQP [9]. Most of the effort in reducing the quantity of work necessary to drive an ITP has been focused on their integration with ATPs. A system that implements such an integration is called a hammer [2]. Hammers generally implement three primary components [2]:

1. **Premise selection:** Select a subset of lemmas from a theory of previously proven lemmas in the ITP that the ATP may use as premises in its proof attempts.
2. **Translation:** Translate the goal into the format specified by the ATP. Commonly this is TPTP (Thousands of Problems for Theorem Provers) format [12].
3. **Reconstruction:** Reconstruct the proof found using the ATP in the ITP.

ACL2 has a preliminary versin of a hammer [6]. The ACL2 hammer implements a translation from ACL2 to TPTP, and implements K-Nearest Neighbors over TF-IDF (Term Frequency - Inverse Document Frequency) representations of lemmas in ACL2 for premise selection. However, the development of a hammer in ACL2 is challenging for a number of language-specific reasons [6]. In this work, we investigate other approaches for simplifying the interaction between ACL2 and its users.

ACL2(ml) is also a tool designed to help ACL2 users [4]. ACL2(ml) is an Emacs extention that gives ACL2 users access to a recurrent clustering algorithm that groups theorems, and function definitions based on the similarity of their hand-crafted features. ACL2 users can use ACL2(ml) to search for theorems and definitions similar to the ones in their immediate environment. The goal of ACL2(ml) differs from the goal of this work. Our models offer precise actions that can directly complete a user's failed proof whereas ACL2(ml) directs users to regions in the ACL2 community "books" (i.e., library files) that might be helpful. Additionally, we train deep-learning models using a supervised learning approach, as opposed to the unsupervised clustering used by ACL2(ml).

Even though deep learning methods have not been applied to problems in the context of ACL2, they have been applied in other ITPs. Specifically, deep learning models have been used for premise selection in the Mizar Mathematical Library [5]. The authors investigate convolutional neural networks (CNNs), long short-term memory networks (LSTMs), and a combination of the two architectures. They also investigate various methods of learning embeddings for lemmas and function definitions as opposed to using hand-crafted features. Though we also investigate deep learning methods for assisting an ITP, our problem is significantly different than premise selection. Also, the deep learning methods we investigate are focused on attention which is distinct from the CNNs and LSTMs.

3 Data Generation

To gather data to train our models, we traverse the ACL2 Community Books, which are public libraries of theorems previously formalized and proved by experts in ACL2. The community books total approximately 9,400 files, 90,000 function definitions, and 170,000 theorems. The data generation tool modifies the theorems in a number of specific ways in order to break them. When this happens, ACL2 reports a set of checkpoints. Each of these checkpoints corresponds to one training example, where the input is the checkpoint and the output is the inverse of the action used to break the theorem. For example, if a theorem was "broken" by removing one of its hypotheses thereby making the theorem too general for ACL2 to prove, or simply false, then each of the checkpoints created by breaking the theorem is associated with replacing the hypothesis.

There are four fundamental ways in which we break theorems when collecting training data: removing a hypothesis from the theorem, removing a hint that was used to prove the theorem, excluding a library that was used to prove the theorem, and removing an individual lemma that was used to prove the theorem. When one of these removals results in a failed proof attempt, the system generates one training example for every reported checkpoint. The most recent data generation run produced almost three million failed proof attempts as seen in Table 1.

3.1 Removing Hypotheses

An obvious way to break a theorem is to remove each of its hypotheses, one at a time. If a hypothesis is actually necessary, ACL2 will fail to prove the modified theorem and the failed proof attempt will result in one or more checkpoints.

Theorem Modification	**# Theorems Failed**
Lemma Removed	1,706,588
Hint Removed	434,206
Hypothesis Removed	421,136
Library Removed	284,866

Table 1. Number of Failed Proof Attempts During Data Collection

The ML system then learns to associate similar checkpoints with the action of adding the removed hypothesis.

For instance, many users new to ACL2 expect the following to be true:

```
(equal (rev (rev x)) x)
```

Here, `rev` computes the reverse of a list. In reality, this property is not true, because `rev` always returns a list, even if its input is not. The correct theorem is given by

```
(implies (true-listp x)
         (equal (rev (rev x)) x))
```

When the hypothesis from this theorem is removed, ACL2 will respond with a checkpoint that looks like

```
(implies (not (consp x))
         (not x))
```

The ACL2 user who tries to prove the original, incorrect theorem will see a similar checkpoint, and the advice tool will suggest adding the `true-listp` hypothesis, helping the user submit the correct theorem.

3.2 Removing Hints

The ACL2 theorem prover provides significant proof automation, but its heuristics are not always sufficient to prove complicated theorems by itself. Thus users commonly provide hints to the theorem prover, either subtly overriding its default search strategy or suggesting new proof strategies. There are, in fact, many different hints that a user can provide, and we currently process only the most common ones. We associate each hint from Table 2 with a separate action-type.

For example, ACL2 is very good at finding induction schemes to prove theorems, but sometimes its heuristics fail to find the right induction scheme. The user can suggest a specific induction scheme, e.g., with induction step $P(n-1, x+n) \rightarrow P(n, x)$, and the ML system is trained to associate this induction scheme with checkpoints that occur when the hint is removed.

We currently simplify the actions associated with some of these hints to create a more manageable task for our models. Without simplification, there would be a large number of actions with only one training example. However, this simplification means that whenever our models recommend adding a hint, the recommendation is less precise and therefore less actionable.

Hint Type	# Checkpoints
Use Hint	246,932
Enable Hint	165,544
Expand Hint	52,661
Disable Hint	42,963
Induct Hint	16,446
By Hint	1,882
Cases Hint	1,765
Nonlinearp Hint	449
Do-not Hint	174

Table 2. Training Pairs By Hint Type

3.3 Removing Books

Another way to break a theorem is to look at the libraries that were imported when it was originally proven. A library is composed of "books" (files) that contain definitions and rules (theorems) that can then be included to help in future proof attempts. For each of the included books, the tool attempts to prove the theorem by excluding rules loaded from that specific book. In many cases, ACL2 is still able to prove the theorem; perhaps that particular book was loaded to help other theorems in the file, or perhaps ACL2 can simply manage to find a different proof. But in many cases, the proof attempt fails after a specific book is removed, which creates one or more training examples.

We have found that this is a very effective way to create advice for users, many of whom may not be fully aware of libraries in the community books that may be effective to reason about some of the same functions they are currently using. For instance, users struggling to prove theorems involving modular arithmetic may be advised to include one of the library books that deal extensively with `mod` and `floor`.

It is worth noting that the advice produced by this technique is inherently easier to handle than the previous two. The reason is that this advice is of the form "load library LIBNAME," which is effectively just a single label that needs to be learned. On the other hand, the previous two techniques produce advice that includes an ACL2 expression, e.g., a hypothesis.

3.4 Removing Definitions and Theorems

The final technique we use to break a theorem is to examine the lemmas (i.e., rules) that ACL2 used to prove the theorem and arrange for each of these lemmas to be removed one at a time. For example, the proof of a theorem may depend crucially on the fact that append is associative, and ACL2 may have been able to use this fact automatically in the original proof, possibly without the user even being aware of this rule. As before, if the rule is necessary for the proof, the system will associate the rule with each checkpoint generated in the failed proof attempt. Note that this type of advice is similar to the previous one in that it appears to be a classification problem to the ML system, e.g., learning just the label “use lemma LEMMANAME” instead of an arbitrary ACL2 expression.

3.5 Partitioning

We call the set of {checkpoint, action} pairs collected by traversing the ACL2 Community Books $\mathcal{E}$. Before training our models, we partition $\mathcal{E}$ into training, validation, and testing sets $\mathcal{E}_{\text{train}}$, $\mathcal{E}_{\text{val}}$, and $\mathcal{E}_{\text{test}}$. We train our models with $\mathcal{E}_{\text{train}}$, tune their hyperparameters with $\mathcal{E}_{\text{val}}$, and evaluate them with $\mathcal{E}_{\text{test}}$. We ensure that all {checkpoint, action} pairs created by breaking a given theorem are in the same set. This ensures that the model is evaluated on a completely unseen set of theorems.

4 Modeling

Given $\mathcal{E}_{\text{train}}$, our goal is to estimate the conditional probability $P(\text{action}|\text{checkpoint})$. An action is composed of an action-type and an action-object. We use an action-type classifier to predict $P(\text{action-type}|\text{checkpoint})$. Then for each action-type, we use an action-object classifier to predict $P(\text{action-object}|\text{action-type, checkpoint})$. We can combine these two classifiers to predict $P(\text{action}|\text{checkpoint})$.

$$\begin{aligned} P(\text{action}|\text{checkpoint}) &= P(\text{action-type} \cap \text{action-object}|\text{checkpoint}) \\ &= P(\text{action-type}|\text{checkpoint}) \cdot P(\text{action-object}|\text{action-type, checkpoint}) \end{aligned} \tag{1}$$

Because there is a separate action-object classifier for each action-type, we refer to each action-object classifier by its action-type. For example, we call the action-object classifier that predicts $P(\text{action-object}|\text{action-type} = \text{use-lemma, checkpoint})$ the “use-lemma classifier”.

To learn these conditional probabilities, we train the action-type model on all examples in $\mathcal{E}_{\text{train}}$ and each action-object model on only the examples in $\mathcal{E}_{\text{train}}$ of the corresponding action-type. The challenge of training such models lies in learning a useful vector representation, or *embedding*, of the checkpoints. Once we have an embedding, we can use standard deep learning to predict the action type or the action object. To learn effective embeddings for checkpoints, we combine components of the transformer architecture as presented in [13], and graph attention networks as presented in [14].

4.1 Preprocessing

Consider the following checkpoint:

```
(implies (and (integerp x) (integerp y)) (integerp (+ x y)))
```

We call this a user-level checkpoint as its presentation is designed for readability by the human user. Internally, an ACL2 checkpoint is a disjunction represented as a list of disjuncts called "literals." In addition, all of the macros in the checkpoint are expanded when creating this internal form; we call this internal form the "translated" version of the checkpoint. The user-level checkpoint above is logically equivalent to the following translated checkpoint:

```
((not (integerp x))
 (not (integerp y))
 (integerp (binary-+ x y)))
```

The translated form of the checkpoint is a more suitable input to our models because it more explicitly embodies the semantics of the checkpoint. We can create models that independently learn an embedding for each literal, and then combine the embeddings to find an embedding for the checkpoint. Furthermore, expanding macros seems desirable since the resulting translated terms have more regular structure (e.g., every function has a fixed arity, whereas this is not true for macros).

Additionally, we normalize the variable names in the translated checkpoints. When we normalize the names, we also indicate the position at which each variable appeared. We perform this normalization because the semantics of a checkpoint should be invariant to the variable names. We reverse the normalization in the actions recommended by the model. In our example, the normalized and translated checkpoint is as follows:

```
((not (integerp var-0)
 (not (integerp var-1))
 (integerp (binary-+ var-0 var-1)))
```

The transformer architecture is known for its ability to encode sequential information. However, checkpoints contain more structural information than sequences. Each literal l_i in a checkpoint can be represented as a tree of symbols. In our example, $l_3 =$ `(integerp (binary-+ var-0 var-1))` can be represented by the tree in Fig. 1. In this tree, internal nodes are function names, and their children are the trees representing their arguments.

Fig. 1. Tree Representation of an ACL2 Literal

$$\begin{tikzcd}
& \texttt{integerp} \arrow[d] & \\
& \texttt{binary-+} \arrow[dl] \arrow[dr] & \\
\texttt{var-0} & & \texttt{var-1}
\end{tikzcd}$$

We cannot directly give a tree to the transformer; we must give it a sequence instead. To create this sequence, we flatten the tree by traversing it breadth-first.

We use breadth-first traversal because for large trees, where we must truncate part of the tree for it to fit as input to the model, we prefer to truncate nodes of greater depth because they tend to be less important to the semantics of the literal. After flattening l_i, we are left with a list of symbols. We perform sub-word tokenization over these symbols using byte-pair encoding as is done in large language models like GPT [10, 11]. We create our byte-pair encoding over the checkpoints from $\mathcal{E}_{\text{train}}$. The result is a list of tokens called the vocabulary of the byte-pair encoding. Let $\mathcal{V}$ be the vocabulary of the byte-pair encoding.

Sub-word tokenization is useful for this problem because it allows us to infer properties of rare symbols in the corpus by transforming them into a list of more common symbols in the corpus. For example, a byte-pair encoding could transform the symbol `var-0` into the tokens `[var, -0<\w>]`. The `<\w>` tag following `-0` indicates the end of a symbol.

After the byte-pair encoding, we are left with a list of lists of tokens. For example, for l_3 we might have:

```
[[int, eger, p<\w>],
 [binary, -+<\w>],
 [var, -0<\w>],
 [var, -1<\w>]]
```

We flatten this list by concatenating the list of tokens for each symbol in l_i. For example, we would transform l_3 into the following list of tokens. When we give these lists as input to our models, we represent each token by its index into $\mathcal{V}$

```
[int, eger, p<\w>, binary, -+<\w>, var, -0<\w>, var, -1<\w>]
```

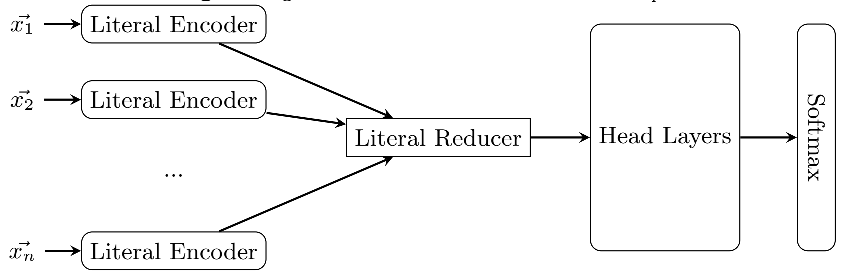
4.2 Architecture

All of our classifiers implement the high level architecture described in Fig. 2. The literal encoder learns an embedding for each literal in a checkpoint. Those embeddings are then combined into an embedding for the checkpoint itself. The embedding for the checkpoint is then fed through a number of head layers, and finally through the softmax activation function which outputs a probability for each class.

Literal Encoder The layout of the literal encoder is based on the encoder component of the transformer architecture [13]. The literal encoder accepts a fixed-length sequence of integers as input, and transforms it into a sequence of embeddings. Let S be the length of the sequence accepted by the literal encoder. Let d_{token} be the length of each embedding in the sequence.

We give the indices of the tokens in l_i as input to the literal encoder. Since the literal encoder only accepts fixed-length sequences, we truncate or pad the list of indices if its length is not S. Let $\vec{x_i}$ be the truncated or padded list of

Fig. 2. High Level Architecture for T and O_p



indices. To transform $\vec{x_i}$ into a sequence of embeddings of its tokens, the literal encoder first creates a one-hot encoding for $\vec{x_i}$ to produce matrix $H_i \in \mathbb{R}^{S \times |\mathcal{V}|}$:

$$H_i[a, b] = 1(\vec{x_i}[a] == b). \tag{2}$$

The literal encoder uses a dense layer to transform H_i into the sequence of embeddings for $\vec{x_i}$. The dense layer uses the weight matrix $W \in \mathbb{R}^{|d_{\text{token}}| \times \mathcal{V}}$, the bias vector $\vec{b}$, and the sigmoid activation function σ. Let $M[i, :]$ be the ith row-vector of matrix M. We call the resulting matrix E_i:

$$E_i[j, :] = \sigma(WH_i[j, :] + \vec{b}) \tag{3}$$

The literal encoder uses a technique called multi head self-attention (MHSA) to learn the interactions between subsets of embeddings in the sequence. It specifies the interactions that MHSA should focus on by using a mask. This technique is similar to the one used in graph attention networks (GATs) which use attention to create embeddings for graphs [14].

For each of H heads in MHSA, three distinct linear transformations are used to project each embedding in the input sequence into a query embedding, a key embedding, and a value embedding. Given input E_i, let Q_i be the sequence of query embeddings, K_i be the sequence of key embeddings, and V_i be the sequence of value embeddings. Let $W_q, W_k, W_v \in \mathbb{R}^{d_{\text{token}} \times d_{\text{head}}}$ be weight matrices. Q_i, K_i, and V_i are calculated with:

$$\begin{aligned} Q_i &= E_i W_q \\ K_i &= E_i W_k \\ V_i &= E_i W_v \end{aligned} \tag{4}$$

Let $M_i \in \{0, 1\}^{S \times S}$ be a binary mask. The output of each head in MHSA is given by matrix A_i:

$$A_i[j, :] = \texttt{softmax}((\frac{Q_i K_i^{\mathrm{T}}}{\sqrt{d}} + -\infty \cdot (1 - M_i))[j, :])^T V_i \tag{5}$$

The matrices resulting from each head of self attention are concatenated together. Each row in the concatenated matrix is fed through a linear transformation into $\mathbb{R}^{d_{\text{token}}}$. The result is a sequence of output embeddings where each of the output embeddings is a weighted combination of the sequence of input embeddings. M_i determines which of the input embeddings can be used in creating each of the output embeddings. Specifically, the kth input embedding can be used to create the jth output embedding only if $M_i[j,k] = 1$. In other words, the jth input embedding can *attend* over the kth input embedding only if $M_i[j,k] = 1$.

The literal encoder performs multiple iterations of MHSA. Each iteration is performed by an encoder layer. An encoder layer also performs transformations other than MHSA. It uses a feed-forward network with two dense layers that performs a nonlinear transformation to each embedding resulting from MHSA. Let d_{ff} be the length of each embedding after the first dense layer. We can tune d_{ff} to control the size of the feed-forward network. The literal encoder also applies layer normalization after MHSA, and after the feed-forward network [1].

The literal encoder has two types of encoder layers. It has symbol layers which create an embedding for each symbol by focusing on the interactions between the tokens of the symbol. Recall each embedding in the sequence of embeddings given to MHSA is a vector-representation of a token in l_i. Let $t(j)$ be the token associated with the jth embedding in the sequence. Also recall that each token belongs to some symbol in l_i. Symbol layers use M_i such that:

$$M_i[j,k] = \begin{cases} 1 & \text{if } t(j) \text{ is the first token of } t(k)\text{'s symbol} \\ 0 & \text{otherwise} \end{cases} \tag{6}$$

The literal encoder does not need to have a symbol layer. If it does have a symbol layer we say that the literal encoder performs symbol-attention. Otherwise, we say that it performs token-attention. If the literal encoder performs symbol-attention, it also needs first-token-mask. Let $\vec{f_i} \in \mathbb{R}^S$ be the first-token-mask for literal l_i. $\vec{f_i}$ is constructed such that:

$$\vec{f_i}[j] = \begin{cases} 1 & \text{if } t(j) \text{ is the first token of its symbol} \\ 0 & \text{otherwise} \end{cases} \tag{7}$$

The literal encoder also has term layers which use the tree structure of the literals to inform which interactions MHSA should focus on. To encode the tree structure of the network, we consider two types of attention:

1. *child-attention*: only allows a node to attend over its children
2. *descendant-attention*: allows a node to attend over all of its descendants.

The problem with child-attention is that it only allows information from a node to propagate up or down the tree as many levels as there are encoder layers. Descendant-attention solves this problem, as the root node directly attends over all leaf nodes. However with descendant-attention, some information about the structure of the tree is lost. The model cannot accurately differentiate whether information is coming from a child, grandchild, or some other descendant. Term layers use M_i such that:

$$M_i[j,k] = \begin{cases} 1 & \text{if } t(k)\text{'s symbol is a descendant of } t(j)\text{'s symbol} \\ 0 & \text{otherwise} \end{cases} \tag{8}$$

In the case of child-attention, the term "descendant" in (8) refers to the direct descendants, or children, of $t(j)$'s symbol. In descendant-attention the term "descendant" refers to any descendant of $t(j)$'s symbol. If the literal encoder is performing symbol-attention, then its mask should only allow attention between the first tokens of any two symbols. We can accomplish this with:

$$M_i = M_i \odot \vec{f}_i \vec{f}_i^T \tag{9}$$

We also need to allow only the information from the first token of a given symbol to advance to the symbol reducer. Let L_i be the sequence of token embeddings passed to the symbol reducer. Let T_i be the sequence of token embeddings after the last term layer. We compute L_i with:

$$L_i[:,j] = \vec{f}_i \odot T_i[:,j] \tag{10}$$

If the literal encoder does not have symbol layers, and is therefore performing term attention, (9) and (10) are not used.

After the last encoder layer in the literal encoder, l_i is represented as a sequence of embeddings, where each embedding is a representation of a token in l_i with encoded information about the token's interactions with the other tokens in l_i. The literal encoder's symbol reducer combines this sequence of token-embeddings into a single literal-embedding by using a linear transformation to project each token-embedding into $\mathbb{R}^{d_{\text{literal}}}$, and then taking the maximum of each column-vector in the resulting matrix [3]. Let L_i be the matrix containing the sequence of embeddings after the last encoder layer. Let $L_i[:,j]$ be the jth column vector of L_i. We combine the token-embeddings into $\vec{l_i}$ with:

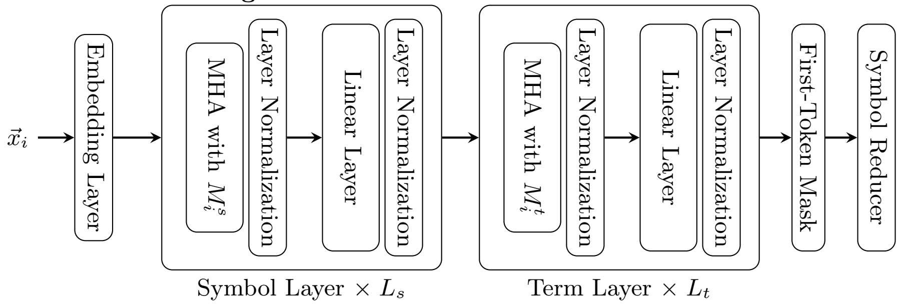
$$\vec{l_i}[j] = \texttt{max}(L_i P[:,j]) \tag{11}$$

The architecture of the complete literal encoder is given in Fig. 3. L_s is the number of symbol layers, and L_t is the number of term layers.

Literal Reducer The literal reducer combines embeddings for literals in the exact same way that the symbol reducer combines embeddings for symbols. The literal reducer projects the literal embeddings into $\mathbb{R}^{d_{\text{checkpoint}}}$ before taking the max of each column.

Head Layers The head layers are a sequence of dense layers. We say that the output of the ith head layer exists in $\mathbb{R}^{d_i}$.

Fig. 3. Architecture of the Literal Encoder



Softmax The softmax activation function is called on the output of the last head layer. It calculates the predicted probability for each class.

4.3 Model Training

We have investigated four methods for encoding structural information by training the action-type models in Table 3.

	token-attention	symbol-attention
child-attention	T_c^t	T_c^s
descendant-attention	T_d^t	T_d^s

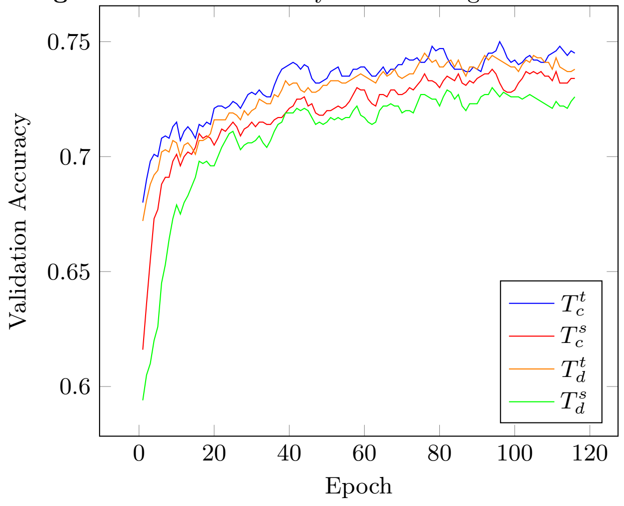
Table 3. Action-Type Model Notation

We train each of these models using the hyperparameters from Table 5 in the appendix. The parameters for all four models are identical other than the presence of a symbol layer in exchange for a term layer in the models that implement symbol-attention.

The results from training each of the four models for 120 epochs are shown in Fig. 4. In this context, an epoch consists taking 400 steps of gradient descent with batches of 64 training examples at a time from $\mathcal{E}_{\text{train}}$. For each model, we calculate the accuracy over 100 batches from $\mathcal{E}_{\text{val}}$ at each epoch. Fig. 4 shows a moving average over 10 epochs for interpretability.

Because T_c^t had the highest accuracy of the four models after 120 epochs, we allowed it to continue training for 400 epochs, and saved its parameter settings after each epoch where it achieved its lowest validation loss. We also trained each action-object model using child and token attention. We used the literal encoder and literal reducer from the action-type model in each action-object model because some action-object models have very few examples to train on. For example, the add-do-not-hint classifier has 156 example in its training set. Using a pre-trained literal encoder and literal reducer means that each action-object classifier only had to train its head layers. This allows each action-object

Fig. 4. Validation Accuracy over Training Iterations



classifier to use information about the properties of checkpoints from other action types. It also permits more efficient training since we can use much larger batch sizes when only training dense layers.

The parameter selection for each action-object classifier is shown in Table 6 of the appendix. When selecting the number of head layers for each model, and d_i for each head layer, we aimed to keep the number of parameters within an order of magnitude of the number of training examples. However, for the use-lemma classifier where most of the parameters come from the the fourth head layer, we found that having a d_3 of 1024 to be more performant than smaller alternatives. We trained each action-object classifier for the number of steps of gradient descent, and batch sizes specified in Table 6 of the appendix. As with T_c^t, we saved the parameter settings after each epoch where the classifier achieves the lowest validation loss.

5 Results

We evaluated each model using $\mathcal{E}_{\text{test}}$. To evaluate the action-type classifier, we randomly selected over 200,000 examples from $\mathcal{E}_{\text{test}}$. We evaluated each action-object classifier on the complete set of examples from $\mathcal{E}_{\text{test}}$ with the classifier's corresponding action-type. When evaluating the models, we use a metric called top-N accuracy. Given classifier Z, top-N accuracy measures the frequency with which the correct class given some input to Z is among the N classes with the highest predicted probabilities from Z. We use top-N accuracy for evaluation because we have developed a tool in ACL2 that can easily try upward of a hundred recommendations. Therefore, there is utility in recommending the correct action to complete a theorem, even if it is not the first recommendation. We show the Top-1, Top-5, and Top-10 accuracies for each model in Table 4. In addition to evaluation metrics, we show the number of examples used to train each model, the number of classes over which each model makes predictions, and the frequency of the most well-represented class.

Classifier	# Train Examples	# Classes	High-Class Freq	Top-1	Top-5	Top 10
action-type	4,312,709	12	69.2	72.5	99.1	100
use-lemma	2,985,134	61,411	3.4	18.5	32.8	38.7
add-library	453,935	2,580	2.8	35.4	61.5	71.1
add-hyp	448,756	13,745	1.9	23.2	35.5	40.4
add-use-hint	197,830	3,164	0.1	32.8	40.3	42.0
add-enable-hint	134,097	4,912	2.5	27.3	41.3	47.6
add-expand-hint	42,300	1,981	15.1	47.7	56.0	59.8
add-disable-hint	34,247	563	11.8	26.5	36.0	39.0
add-induct-hint	13,015	464	6.7	22.3	28.9	32.5
add-by-hint	1,473	683	1.7	18.1	26.2	26.9
add-cases-hint	1,407	919	4.0	14.6	27.8	32.6
add-do-not-hint	156	5	52.5	100.0	100.0	-

Table 4. Individual Model Evaluation

To simulate a more realistic evaluation of our models, we perform a combined evaluation in which we treat all of our models, as one classifier, C_{combined} which models $P(\text{action}|\text{checkpoint})$ as is shown in (1). We then estimate the top-N accuracy of C_{combined} where $N \in 1, 2, ..., 100$ using 128,000 randomly selected examples from $\mathcal{E}_{\text{test}}$. The results of this evaluation are plotted in Fig. 5.

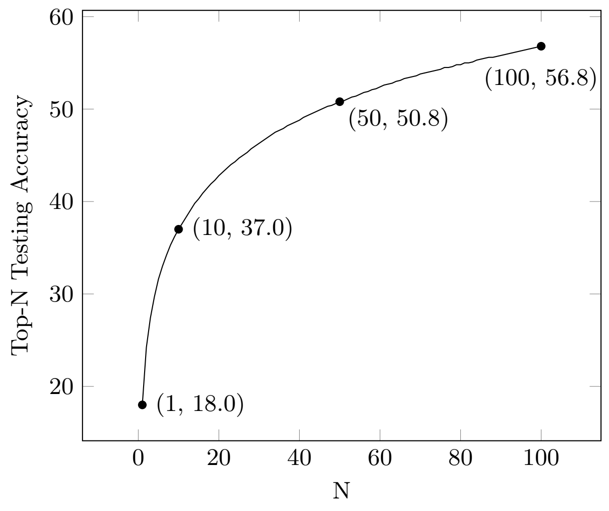
We can interpret the top-N accuracy of C_{combined} as the frequency in which C_{combined} will propose an action, a within the top N recommendations given a checkpoint from the failed proof attempt of a theorem, t, where applying a to t yields a theorem that ACL2 can prove. This evaluation only holds for theorems where our data generation tool could possibly generate the broken theorem t from its successful counterpart.

Note that there exists some broken theorems in that can be completed by applying multiple actions. Therefore, our estimates for the top-N accuracy of C_{combined} are lower bounds because they assume that there is only one action that completes the theorem.

6 Conclusion

We train deep learning models based on attention to generate proof advice for ACL2 users encountering failed proof attempts. We train and evaluate these models using {checkpoint, action} pairs generated by breaking theorems in the ACL2 community books. We compare various methods of using masked attention to encode checkpoints. Advice from our models is available through a web-interface, and through a public ACL2 library. We created this advice tool in order to help all ACL2 users become more effective at proving theorems.

Fig. 5. Combined Model Evaluation



6.1 Future Work

We plan to develop a system of directly evaluating our models' proof advice in ACL2. Currently we can only interpret the modeling results in the context of our data-generation procedure. We also only know if the advice is correct if it exactly matches the action from the {checkpoint, action } pair. However, an evaluation of the advice in ACL2 would give a more realistic measure of the models' performance.

Evaluation of the proof advice in ACL2 would also give us the ability to compare the performance of the classifiers presented in this work to generative proof advice models. We suspect that our method for evaluating the results of our classifiers is inadequate for generative approaches since they are less likely to produce actions that exactly match the action given in the testing set.

Appendix Parameter Settings

Parameter	T_c^t	T_c^s	T_d^t	T_d^s
L_s	0	1	0	1
L_t	2	1	2	1
S	256	256	256	256
H	4	4	4	4
d_{token}	256	256	256	256
d_{ff}	256	256	256	256
d_{literal}	2048	2048	2048	2048
$d_{\text{checkpoint}}$	2048	2048	2048	2048
d_1	32	32	32	32
d_2	12	12	12	12

Table 5. Parameters of T

Classifier	d_1	d_2	d_3	d_4	Parameters	Batch Size	Train Steps
use-lemma	1,024	1,024	1,024	61,411	67,143,651	2048	20,000
add-library	256	256	32	2,580	683,700	2048	10,000
add-hyp	256	256	32	13,745	1,117,937	2048	10,000
add-use-hint	128	128	32	3,164	387,324	2048	10,000
add-enable-hint	128	128	32	4,912	445,008	2048	10,000
add-expand-hint	64	64	1,981	-	264,061	2048	1,000
add-disable-hint	128	64	64	563	311,283	2048	1,000
add-induct-hint	32	32	464	-	81,936	2048	1,000
add-by-hint	32	32	683	-	89,163	1024	1,000
add-cases-hint	16	16	919	-	48,679	1024	1,000
add-do-not-hint	5	-	-	-	10,245	128	500

Table 6. Parameters for each Action Object Classifier

References

1. Ba, J.L., Kiros, J.R., Hinton, G.E.: Layer normalization. arXiv preprint arXiv:1607.06450 (2016)
2. Blanchette, J.C., Kaliszyk, C., Paulson, L.C., Urban, J.: Hammering towards qed. Journal of Formalized Reasoning **9**(1), 101–148 (2016)
3. Hamilton, W., Ying, Z., Leskovec, J.: Inductive representation learning on large graphs. Advances in neural information processing systems **30** (2017)
4. Heras, J., Komendantskaya, E.: Acl2 (ml): Machine-learning for acl2. arXiv preprint arXiv:1404.3034 (2014)
5. Irving, G., Szegedy, C., Alemi, A.A., Eén, N., Chollet, F., Urban, J.: Deepmath-deep sequence models for premise selection. Advances in neural information processing systems **29** (2016)
6. Joosten, S., Kaliszyk, C., Urban, J.: Initial experiments with TPTP-style automated theorem provers on ACL2 problems. arXiv preprint arXiv:1406.1559 (2014)
7. Kaufmann, M., Manolios, P., J S. Moore: Computer-Aided Reasoning: An Approach. Kluwer Academic Press, Boston, MA. (2000). https://doi.org/10.1007/978-1-4615-4449-4
8. Kühlwein, D.: Machine Learning for Automated Reasoning. [Sl: sn] (2014)
9. McCune, W.: Solution of the robbins problem. Journal of Automated Reasoning **19**(3), 263–276 (1997)
10. Radford, A., Narasimhan, K., Salimans, T., Sutskever, I., et al.: Improving language understanding by generative pre-training (2018)
11. Sennrich, R., Haddow, B., Birch, A.: Neural machine translation of rare words with subword units. arXiv preprint arXiv:1508.07909 (2015)
12. Sutcliffe, G.: The TPTP World–infrastructure for automated reasoning. In: International Conference on Logic for Programming Artificial Intelligence and Reasoning. pp. 1–12. Springer (2010)
13. Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A.N., Kaiser, Ł., Polosukhin, I.: Attention is all you need. Advances in neural information processing systems **30** (2017)
14. Veličković, P., Cucurull, G., Casanova, A., Romero, A., Lio, P., Bengio, Y.: Graph attention networks. arXiv preprint arXiv:1710.10903 (2017)