ProverBot9000: Neural Networks for Proof Assistance

Joseph Redmon and Alex Sanchez-Stern

Abstract

We introduce ProverBot9000, a state-of-the-art tool for proof automation and assistance. ProverBot9000 examines partially finished Coq proofs and proposes tactics to make progress on the proof. It generates these tactics using a neural network-based language model of Ltac. ProverBot9000 is trained on human-generated proofs so it suggests tactics that human experts are likely to use in a given proof state. Furthermore, it can be fine-tuned for a specific domain (e.g. distributed systems or compilers) simply by adding completed proofs in that domain to its training set. We evaluate ProverBot9000 on proofs of peephole optimization correctness in a verified C compiler.

1 Introduction

Software developers working on large scale, sensitive systems face a daunting challenge. Billions of dollars and thousands of lives depend on the correctness and stability of their code. At NASA, developers write tens of thousands of pages of specification for a new project before a single line of code is written. It takes an average of 2 years for a new feature to go from proposal to implementation in the space shuttle program. At Boeing, software developers read through printouts of the code and check it against specifications by hand.

Formal verification offers an enticing alternative to this model of development. In Coq or Dafny, program specifications are concise while machine-checked proofs guarantee that an implementation follows the specification. Verification replaces the need for large-scale testing efforts or manual code reviews. Developers can safely add features and the verification engine will check if they have unintended consequences.

Researchers already use formal verification for building compilers, optimizers, distributed systems, and control software. However, industrial programmers still rely on large-scale testing over formal methods. Writing proofs about programs, especially large systems, requires expert-level knowledge both in that system domain and in type theory and constructive logic.

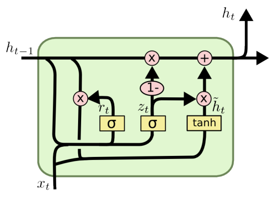
ProverBot9000 is a proof assistant assistant designed to bridge this knowledge gap. It learns proof engineering in Coq by analyzing thousands of lines of expert generated proofs. At test time, it helps novice proof engineers by suggesting possible tactics to make progress in a proof. It can also run in search mode where it generates and evaluates possible tactics to complete proofs on its own.

2 The Model

To prove a statement in Coq, programmers apply tactics to a proof state that either modify the proof's goal (what you need to show to finish the proof), or the proof's context (statements you have already proven or assumed to be true). The tactic may depend on what tactics have been already run, or what tactics or theorems are defined earlier in the file or included files. Thus, we model proof engineering as a function from proof goals G, contexts C, previous tactics P, and prior definitions D, to tactics T:

$$(G \times C \times P \times D) \to T$$

We use a recurrent neural network (RNN) to learn an approximation of this function from expert-generated training samples. RNNs achieve state-of-the-art performance on language modelling tasks. Our model uses Gated Recurrent Units (GRUs), a type of neural network layer that models time series data. GRUs have a memory state that they update at every time step based on the previous time step and the current input to the layer.



$$z_t = \sigma\left(W_z \cdot [h_{t-1}, x_t]\right)$$
$$r_t = \sigma\left(W_r \cdot [h_{t-1}, x_t]\right)$$
$$\tilde{h}_t = \tanh\left(W \cdot [r_t * h_{t-1}, x_t]\right)$$
$$h_t = (1 - z_t) * h_{t-1} + z_t * \tilde{h}_t$$

Figure 1: GRU Layer from `http://colah.github.io/posts/2015-08-Understanding-LSTMs/`

We use a character-level language model because it can learn more general representations and adapts better between tasks (say, fine-tuning for a specific domain).

Character-level language models do not tokenize the text they are modelling but instead learn to model the text as a stream of characters. There are a number of pros and cons to this approach. For example, character-level RNN's have to learn to spell, and . However, character-level RNN's can learn similarities between terms based on their spelling (for instance `intro` and `intros`) while a token-level RNN would just see the terms as two distinct tokens with no relationship.

3 Dataset

To model expert-level proof engineering we need expert-generated proofs! We mine our training data from the verified C compiler CompCert and the verified peephole optimizer built on top of CompCert, Peek. We script `coqtop` to run the proofs and extract the proof context and goals from partially completed proofs. This gives us 94,000 proof states and subsequent tactics for training. We leave out two proof files for testing, the Peek proofs of Aiken5 and Aiken6. This gives us 145 test exmaples.

4 Results

As a simple baseline, we train an RNN on only proof goals and tactics in their original form. This model can only see the current goal, not the proof context or the rest of the file, when predicting the next tactic to use.

We train a 4-layer network that uses 3 GRU layers of 1024 neurons, and one fully connected layer to output our final prediction. The input and output is 256 units since we are modelling character streams. We train the model for 8 epochs on our training data using stochastic gradient descent with a batch size of 128 and a time step size of 256 (gradients are only backpropagated 256 steps into the past).

First we measure perplexity on the test set. Perplexity measures how well the model fits the test data. We use the standard NLP perplexity formula:

$$2^{-1/N \sum_i \log q(x_i)}$$

Where N is the number of examples, and $q(x_i)$ is the probability assigned to example x_i by the model.

This baseline achieves a character-level perplexity on the test set of 1.65 and a tactic-level perplexity of 15959.64. While our character-level perplexity is encouraging, the tactic-level perplexity is especially high, mostly due to the combinatorial structure of tactics. Our model does well on simple, one-line tactics like `intros.` but does much worse on longer tactics like:

```
P0 _clear step_through;
   P0 _clear at_code;
   P0 _clear at_code_end;
   P0 _clear not_after_label_in_code;
   P0 _clear st_rs;
   inv_state;
   P0 bump val_eq;
   P0 bump exec_instr_bits.
```

Since tactics can be strung together like this in arbitrary ways our model can't accurately predict all the possible combinations. One option to deal with this is to break these tactics up by applying the components individually to each subgoal instead of using semicolons.

Even this basic model achieves some success. On the test set our model generates the correct tactic 11.0% of the time. Some success cases are:

```
Goal:
concrete = fnd aiken_6_defs ->
    StepEquiv.rewrite

Tactic:
intros.



Goal:
(forall reg : preg,
 PC = reg \/ r2 = reg \/ False ->
 val_eq
   (nextinstr_nf
      (nextinstr_nf r3 # r1 <- (Val.sub (r3 r1) Vone)) # r2 <-
      (Val.and (nextinstr_nf r3 # r1 <- (Val.sub (r3 r1) Vone) r2)
         (nextinstr_nf r3 # r1 <- (Val.sub (r3 r1) Vone) r1)) reg)
   (nextinstr (nextinstr (nextinstr rsr # r1 <- (Vint zero))) reg))
   /MemEq.mem_eq a ml x4

Tactic:
split.



Goal:
StepEquiv.step_through_equiv_live (fnd aiken_5_defs)
  (rpl aiken_5_defs) (lv_in aiken_5_defs) (lv_out aiken_5_defs)

Tactic:
prep_l.



Goal:
val_eq
  (Val.and (nextinstr rsl # r1 <- (Val.of_optbool None) r2)
     (Val.sub (nextinstr rsl # r1 <- (Val.of_optbool None) r1) Vone))
  (nextinstr (nextinstr rsr # r1 <- (Vint zero)) r2)

Tactic:
preg_simpl.



Goal:
val_eq (Val.add (Val.add (Val.add (rsl PC) Vone) Vone) Vone)
  (Val.add (Val.add (Val.add (rsr PC) Vone) Vone) Vone)

Tactic:
repeat find_rewrite_goal.
```

Some of the failures are only a few characters different while others are way off base:

```
Goal:
Vint (Int.and i0 Int.mone) = rsr r2

Truth:
rewrite <- H0.

Prediction:
repeat break_match_hyp; try congruence.
```

```
Goal:
exists (rsr' : regset) (mr' : Memory.Mem.mem),
  step_through z (Pmov_ri r1 seven :: Psub_rr r1 r2 :: Pnop :: nil)
    (env tprog) (State_bits rsr mr md') Events.E0
    (State_bits rsr' mr' md') /  (forall reg : preg,
   PC = reg \/ r2 = reg \/ False -> val_eq (rsl' reg) (rsr' reg)) /  MemEq.mem_eq [illegible] mr'

Truth:
do 3 step_r.

Prediction:
step_r.



Goal:
val_eq (Val.and (rsl r2) Vundef) Vundef

Truth:
unfold Val.and.

Prediction:
unfold Val.loword.
```

To improve the performance of Proverbot9000, we break down the larger, composed tactics into single atomic tactics, with goals in between. We have a heuristic process which rearranges and duplicates tactics which are composed via semi-colon to produce more straightforward proofs which operate on each proof branch independently. Because existential variables in Coq can cause cross-effects between branches of a proof, it is not always possible to do this well, and in some cases we have to leave semi-colons in the proofs. But we were able to remove them from the majority of our data. Once we did so, we were able to get much better performance from Proverbot9000.

We trained Proverbot9000 on three different data sets, to determine which information was most useful for generating new tactics:

1. Just the goal to be proven
2. Just the three most recent tactics in the proof
3. The goal to be proved, followed by the most recent tactic in the proof

Proverbot9000 was able to produce a tactic identical to the one written by a human in 21.1% of cases when it was given just the goal to be proven. When given the three most recent tactics, it was only able to predict the next tactic in 8.5% of cases. When given first the goal, and then the most recent tactic, it was successful in 16% of cases.

It may seem surprising that the accuracy was lower when given the goal and the previous tactic than it was when given just the goal. But here the ordering is important. In this setup, the goal was farther back in Proverbot9000's memory than it was when it was given just the goal. As a result, it informed the internal state less, and produced weaker results.

These results seem to indicate that the goal is very useful in determining which tactic should be executed next, while previous tactics are less informative. The proof context and information about other names in scope probably also carries a lot of information about what tactic should next be executed, but we have not yet been able to find a good way to integrate such large bodies of text into our character based model.

It's important to note that these results measure how well Proverbot9000 could duplicate exactly the tactics executed by a human, not how well it could advance the proof. It's often the case that there are multiple ways to solve a proof, some of them very similar, that would all work equally well. It may be that Proverbot9000 sometimes predicts a tactic that, while not the one the human had chosen next, works equally well, and may have been chosen by a different human. Unfortunately this is hard to evaluate, since we have not yet attempted to generate entire proofs with Proverbot9000, and it is difficult to say when a proof has "progressed."

5 Future Work

We believe the results thus far in predicting tactics with Proverbot9000 have only scratched the surface of what is possible in this space. We hope to in the future be able to add much more structure and domain knowledge into our model, and vastly improve our results. In particular, we'd like to design a multi-stream RNN where a separate input stream is processed for the context, goal, and global (file) context and fed into a single output RNN that predicts the tactic. Or we may find some other tricky way of modelling these 3 disparate components. We also may improve our character based model to integrate tokenization to some extent which mitigates the burden of learning to spell, and provides improved memory as we can fit more information into fewer tokens.

We would also like to evaluate ProverBot9000 in standalone mode where it searches for full proofs on it's own, and responds to feedback from the Coq prover process. One could imagine a Proverbot9000 which tries one tactic, and if it receives an error from `coqtop`, backtracks and tries another.