Lean Conjecture Generator

Matt Feller

August 2024

1 Problem statement

In this project, we train a generative AI large-language model which conjectures lemmas that may be useful while writing a proof in the theorem-prover Lean. Such a tool could be helpful for human Lean users, but also to incorporate conjectural lemmas into an automatic proof-search program like LeanDojo's ReProver.

2 Background

Although LLMs such as ChatGPT have recently become popular for generating code, the output tends to require substantial checking and debugging by humans. In general, generating reliable code is still out of reach for current AI models. Interactive theorem provers such as Lean provide an ideal setting for exploring improvements in generating code, in part because of the freely available repository of proofs from the efforts at formalizing math (specifically, mathlib4 in Lean 4), but also for the simple fact that theorem prover code verifies itself.

Lean, as an interactive theorem prover, is both a programming language and a setting for writing proofs. One can define variables, functions, and such just like an ordinary programming language, but its primary use is to write formal theorem statements and provide formal proofs. When writing a proof, a second window tracks the current proof state, keeping track of what is known/assumed and what is left to be proved. With every new line of code added to the proof, the proof state is updated. The user adds lines of code until the goal state is reached without errors, at which point the theorem has been formally proved.

There are a number Lean theorem-proving AI models out there, but none achieves a success rate above 42% on the miniF2F benchmark, with the number being even lower for models that do not use reinforcement learning. One of the best and most recent non-reinforcement learning models is LeanDojo's open-source ReProver, which achieves 26.5% success on miniF2F.

Even the best of these automatic theorem-proving algorithms uses a heavy amount of guess-and-check, meaning that they tend to work effectively to recreate shorter proofs, but struggle to piece together longer, more complex arguments.

To create AI capable of efficiently producing long, complex proofs, we would want to allow the system to break down the problem into smaller pieces, in the way that a human mathematician attempts a complex problem. Given a claim one would like to prove, a common approach is to identify a number of simpler claims from which the result can be proved, then proving those simpler claims one by one. The aim of this project is to create a generative AI model which produces simpler claims like this, with the idea that it can be incorporated into a proof-search algorithm following the above outline.

In order to train our model, we leverage the open-source work behind LeanDojo's ReProver model in two ways: we use their benchmark data with a challenging train/test split to train our model, and we use their tokenizer and their ByT5 model available on HuggingFace as a starting point for our training in order to gain a head-start via transferred learning.

3 Data wrangling and exploratory analysis

The data comes from this dataset. We only use the corpus and the data in the novel_premises folder. This dataset contains over 100,000 theorems and detailed information about each step of their proofs. The most relevant information for our purposes is:

- The proof state before running the next tactic (line of code), stored as a string containing the current assumptions and the remaining goals.
- The next tactic (line of code), with annotation marks ¡a¿ and ¡/a¿ around the names of other theorems referenced by the tactic.
- Information about the location of the referenced theorems within the dataset. (The names alone are often not unique indicators.)

Because our goal is to have our model predict tactics which reference possibly new and unproven conjectures instead of known results, we add an addition column to this data which is the next tactic used, but with the names of theorems replaced with the content of the theorems. We do so with a Python function that finds each referenced theorem in the corpus, where the content of the theorem is also stored, then replaces the name with the content.

Now having "expanded tactics" for each data entry, we use the proof states as our inputs and the expanded tactics as our outputs for training the model.

4 Preprocessing

We start with the sequence-to-sequence ByT5 transformer model from LeanDojo's ReProver found here. Although this is a ByT5 model, meaning that the tokenization is applied to individual characters, we benefit from transferred learning by starting with a model which already mimics Lean's syntax well.

In order to train using this model, we extract the data into a list of inputs and outputs, and transform these into PyTorch tensors of uniform length. Because the length of many of the inputs and outputs is relatively long, up to 2000 characters or more, we set a max token length of 1400 initially, although for later fine-tuning we limit ourselves to just the shortest 20000 data points, reducing the max token length to 140.

The dataset came split into training, validation, and test sets already. We limit the training data only to proof steps where the tactic used had some content put into its ”expanded tactic”, which are indicated by the presence of ¡c¿ ¡/c¿ markers. There were about 137,000 such input/output pairs used in training for the first 9 epochs, then 50 epochs of the shortest 20,000 or so (which could train much faster due to the reduced dimensionality). The validation and test sets contain thousands of entries as well, although for much of the testing we used a random sample of a few hundred due to computational limitations.

5 Modeling

To track the model’s performance during training, we use BLEU and Chr f-scores, which are standard metrics for evaluating sequence-to-sequence models for applications like translation between languages.

Figure 1: BLEU scores

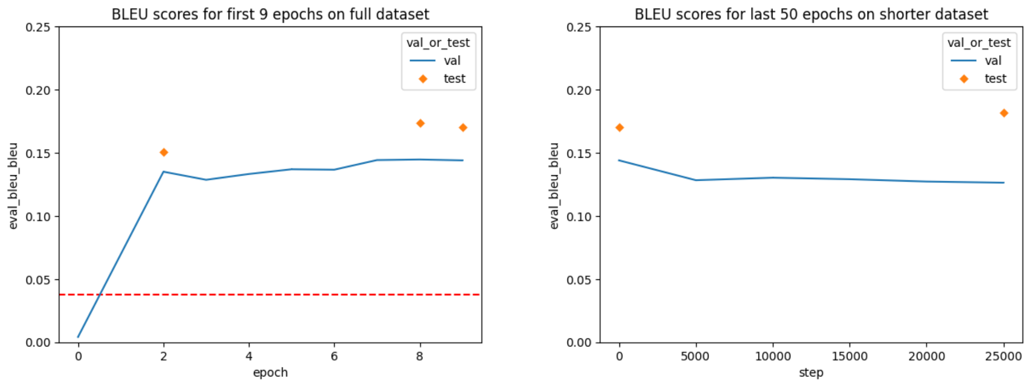
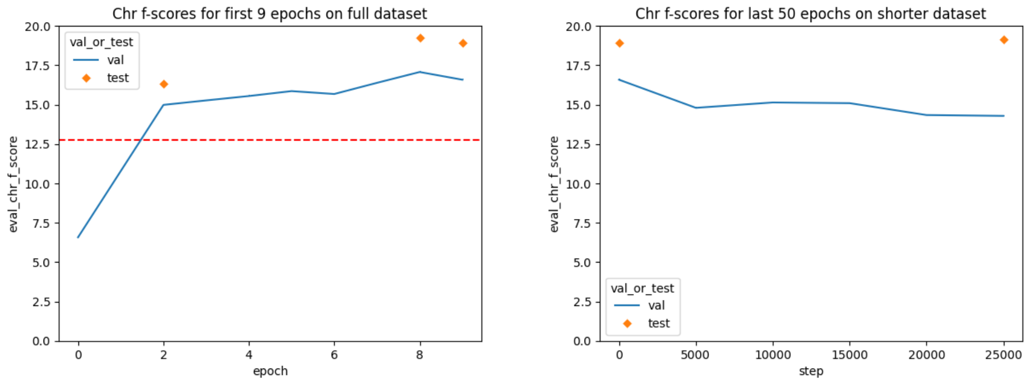


Figure 2: Chr f-scores



The dashed red line in the graphs above indicates the scores of LeanDojo’s ReProver model on their training set. Although it isn’t necessarily meaningful to compare the scores of sequence-to-sequence models with different functions, the fact that our model achieves higher scores does provide a level of confidence that the training has been fruitful.

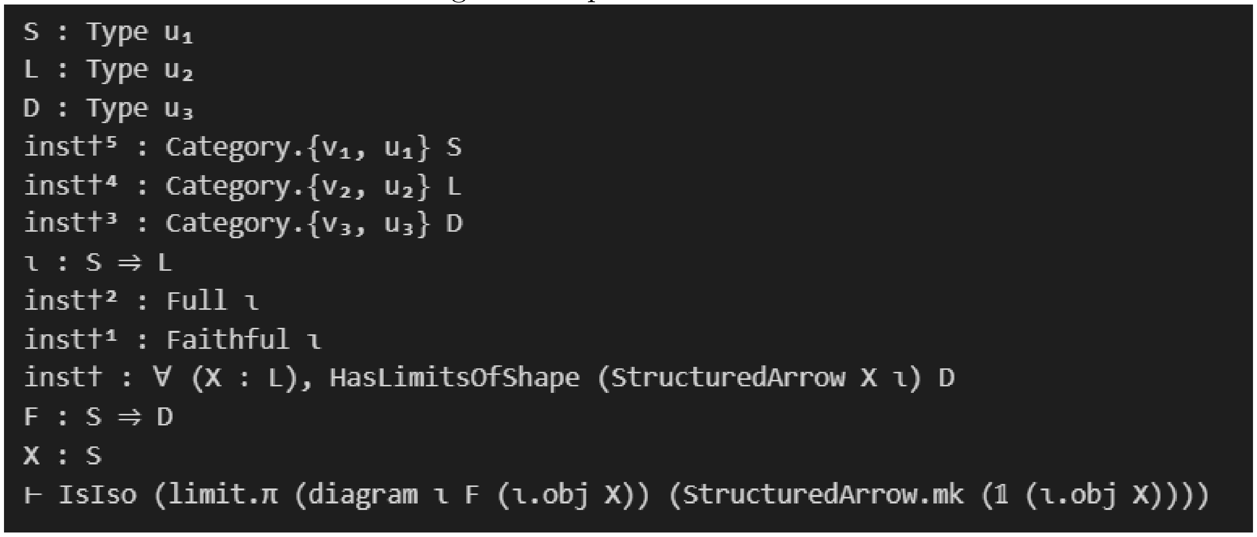
Based on the validation scores alone, the most successful models appear to be the epoch 8 and 9 models, since the validation scores decline after training on the shorter dataset. However, we see that when evaluated on the test set, there are better performances overall and no apparent decline after further training.

Although these scores are encouraging, they do not tell the whole story behind the model's performance, so we do some more hands-on testing and examine the results.

6 Testing

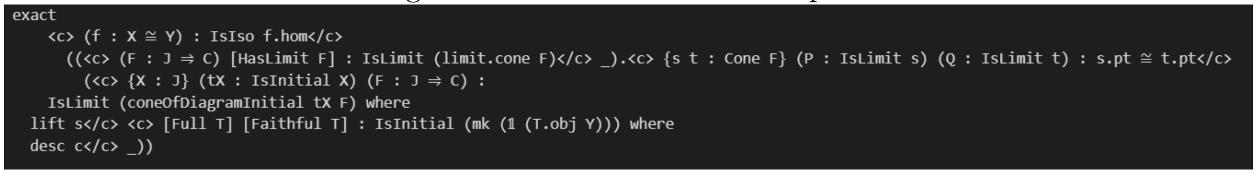
Let's examine the predictions from three of our models in depth: after 2 full-data epochs ('model 0'), after 9 full-data epochs ('model 1'), and after 50 additional shorter-data epochs ('model 2').

Figure 3: Input: 'state_before'



```
S : Type u₁
L : Type u₂
D : Type u₃
inst†⁵ : Category.{v₁, u₁} S
inst†⁴ : Category.{v₂, u₂} L
inst†³ : Category.{v₃, u₃} D
ι : S ⥤ L
inst†² : Full ι
inst†¹ : Faithful ι
inst† : ∀ (X : L), HasLimitsOfShape (StructuredArrow X ι) D
F : S ⥤ D
X : S
⊢ IsIso (limit.π (diagram ι F (ι.obj X)) (StructuredArrow.mk (𝟙 (ι.obj X))))
```

Figure 4: Actual tactic used in proof



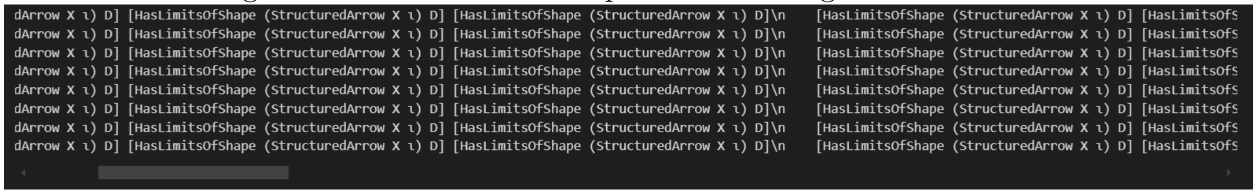
```
exact
    <c> (f : X ≅ Y) : IsIso f.hom</c>
      ((<c> (F : J ⥤ C) [HasLimit F] : IsLimit (limit.cone F)</c> _).<c> {s t : Cone F} (P : IsLimit s) (Q : IsLimit t) : s.pt ≅ t.pt</c>
        (<c> {X : J} (tX : IsInitial X) (F : J ⥤ C) :
    IsLimit (coneOfDiagramInitial tX F) where
  lift s</c> <c> [Full T] [Faithful T] : IsInitial (mk (𝟙 (T.obj Y))) where
  desc c</c> _))
```

Figure 3 shows a random proof state from the test set, and Figure 4 shows the tactic used in the actual proof.

The predictions from these three models are shown in Figures 5, 6, and 7.

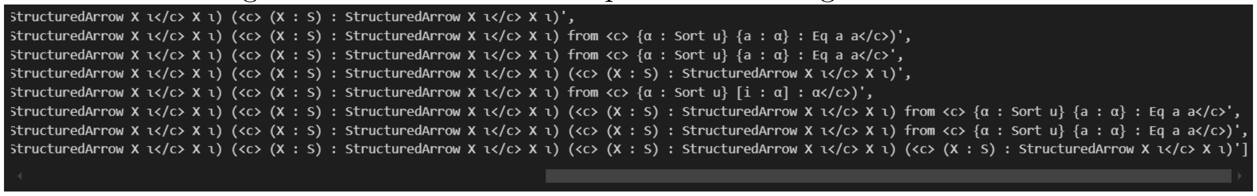
We see that while model 0's metrics above were not too different from those of model 1 and 2, its output is not very useful due to excessive repetition. Model 0 has similar difficulty with other inputs. The output for model 1 has some repetition for this prompt as well, but

Figure 5: Model 0: After 2 epochs of training on full dataset



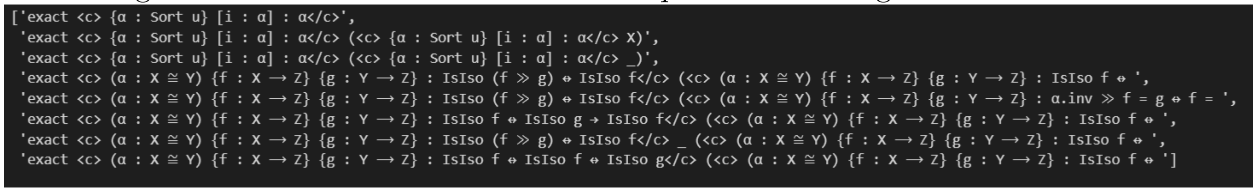
```
dArrow X ι) D] [HasLimitsOfShape (StructuredArrow X ι) D] [HasLimitsOfShape (StructuredArrow X ι) D]\n    [HasLimitsOfShape (StructuredArrow X ι) D] [HasLimitsOfS
dArrow X ι) D] [HasLimitsOfShape (StructuredArrow X ι) D] [HasLimitsOfShape (StructuredArrow X ι) D]\n    [HasLimitsOfShape (StructuredArrow X ι) D] [HasLimitsOfS
dArrow X ι) D] [HasLimitsOfShape (StructuredArrow X ι) D] [HasLimitsOfShape (StructuredArrow X ι) D]\n    [HasLimitsOfShape (StructuredArrow X ι) D] [HasLimitsOfS
dArrow X ι) D] [HasLimitsOfShape (StructuredArrow X ι) D] [HasLimitsOfShape (StructuredArrow X ι) D]\n    [HasLimitsOfShape (StructuredArrow X ι) D] [HasLimitsOfS
dArrow X ι) D] [HasLimitsOfShape (StructuredArrow X ι) D] [HasLimitsOfShape (StructuredArrow X ι) D]\n    [HasLimitsOfShape (StructuredArrow X ι) D] [HasLimitsOfS
dArrow X ι) D] [HasLimitsOfShape (StructuredArrow X ι) D] [HasLimitsOfShape (StructuredArrow X ι) D]\n    [HasLimitsOfShape (StructuredArrow X ι) D] [HasLimitsOfS
dArrow X ι) D] [HasLimitsOfShape (StructuredArrow X ι) D] [HasLimitsOfShape (StructuredArrow X ι) D]\n    [HasLimitsOfShape (StructuredArrow X ι) D] [HasLimitsOfS
dArrow X ι) D] [HasLimitsOfShape (StructuredArrow X ι) D] [HasLimitsOfShape (StructuredArrow X ι) D]\n    [HasLimitsOfShape (StructuredArrow X ι) D] [HasLimitsOfS
```

Figure 6: Model 1: After 9 epochs of training on full dataset



```
StructuredArrow X ι</c> X ι) (<c> (X : S) : StructuredArrow X ι</c> X ι)',
StructuredArrow X ι</c> X ι) (<c> (X : S) : StructuredArrow X ι</c> X ι) from <c> {α : Sort u} {a : α} : Eq a a</c>)',
StructuredArrow X ι</c> X ι) (<c> (X : S) : StructuredArrow X ι</c> X ι) from <c> {α : Sort u} {a : α} : Eq a a</c>',
StructuredArrow X ι</c> X ι) (<c> (X : S) : StructuredArrow X ι</c> X ι) (<c> (X : S) : StructuredArrow X ι</c> X ι)',
StructuredArrow X ι</c> X ι) (<c> (X : S) : StructuredArrow X ι</c> X ι) from <c> {α : Sort u} [i : α] : α</c>)',
StructuredArrow X ι</c> X ι) (<c> (X : S) : StructuredArrow X ι</c> X ι) (<c> (X : S) : StructuredArrow X ι</c> X ι) from <c> {α : Sort u} {a : α} : Eq a a</c>',
StructuredArrow X ι</c> X ι) (<c> (X : S) : StructuredArrow X ι</c> X ι) (<c> (X : S) : StructuredArrow X ι</c> X ι) from <c> {α : Sort u} {a : α} : Eq a a</c>)',
StructuredArrow X ι</c> X ι) (<c> (X : S) : StructuredArrow X ι</c> X ι) (<c> (X : S) : StructuredArrow X ι</c> X ι) (<c> (X : S) : StructuredArrow X ι</c> X ι)']
```

Figure 7: Model 2: After 50 additional epochs of training on shorter dataset



```
['exact <c> {α : Sort u} [i : α] : α</c>',
 'exact <c> {α : Sort u} [i : α] : α</c> (<c> {α : Sort u} [i : α] : α</c> X)',
 'exact <c> {α : Sort u} [i : α] : α</c> (<c> {α : Sort u} [i : α] : α</c> _)',
 'exact <c> (α : X ≅ Y) {f : X ⟶ Z} {g : Y ⟶ Z} : IsIso (f ≫ g) ↔ IsIso f</c> (<c> (α : X ≅ Y) {f : X ⟶ Z} {g : Y ⟶ Z} : IsIso f ↔ ',
 'exact <c> (α : X ≅ Y) {f : X ⟶ Z} {g : Y ⟶ Z} : IsIso (f ≫ g) ↔ IsIso f</c> (<c> (α : X ≅ Y) {f : X ⟶ Z} {g : Y ⟶ Z} : α.inv ≫ f = g ↔ f = ',
 'exact <c> (α : X ≅ Y) {f : X ⟶ Z} {g : Y ⟶ Z} : IsIso f ↔ IsIso g → IsIso f</c> (<c> (α : X ≅ Y) {f : X ⟶ Z} {g : Y ⟶ Z} : IsIso f ↔ ',
 'exact <c> (α : X ≅ Y) {f : X ⟶ Z} {g : Y ⟶ Z} : IsIso (f ≫ g) ↔ IsIso f</c> _ (<c> (α : X ≅ Y) {f : X ⟶ Z} {g : Y ⟶ Z} : IsIso f ↔ ',
 'exact <c> (α : X ≅ Y) {f : X ⟶ Z} {g : Y ⟶ Z} : IsIso f ↔ IsIso f ↔ IsIso g</c> (<c> (α : X ≅ Y) {f : X ⟶ Z} {g : Y ⟶ Z} : IsIso f ↔ ']
```

still within the realm of reason, and is not as prone to it for other prompts. Model 2 has substantially different output, while also being the shortest and least repetitive. Both models 1 and 2 have the type of output we hoped to see from our model, so either is a viable choice for implementing into a copilot or automated prover system. In fact, since their outputs are so different, they could be used effectively in tandem, offering alternative guesses of a different flavor.

7 Conclusion

At first, while training, it appeared that performance on the validation set was diminishing with further training, apparently starting to overfit to the shorter and smaller dataset. However, upon further testing it became clear that the model had similar or better performance after this additional training, while being substantially different the model trained only on the full dataset.

The conclusion to take from this analysis is that the metrics available are not sufficient for quantifying the performance we would like to see in order to guide further fine-tuning. We saw similar BLEU scores and Chr f-scores but vastly different behavior from all of the models after 2 epochs of training.

In order to proceed rigorously with more training, we need a good way to evaluate the model based on whether the code it generates runs in Lean. Although this fine-tuning would be beneficial, but it relies upon interaction with Lean via LeanDojo or a similar system. Incorporating an automatic Lean interaction framework like that has turned out to be a technical challenge beyond the scope of this project.

The main steps for future work are:

- Train using a loss function that penalizes the model for code which does not run in Lean. (Potentially intricate and computationally-intensive to implement.)
- Train using a loss function that rewards the model for outputting conjectures which help solve the ultimate problem. (Probably even more computationally-intensive, maybe beyond what is feasible.)
- Create a copilot plug-in for Lean which suggests the model's conjectures if above a certain confidence threshold.
- Adjust LeanDojo's ReProver algorithm to allow for using and proving conjectures made by our model. Experiment with various meta-variables in the algorithm, such as a confidence threshold determining which conjectures to explore, how many conjectures to allow, and controls on recursion depth.

The ultimate hope is that by incorporating one of our models, or a further fine-tuned version, into the ReProver algorithm, the resulting AI prover will be better equipped to produce novel an complex proofs better than the existing AI provers. Although implementing this idea is outside the scope of this current project, the model we have produced is a viable candidate for being implemented in this way, and in their current form can effectively function as an aid to human Lean users.