Contributions to Neural Theorem Proving

by

Jesse Michael Han

B.Sc., University of California, Los Angeles, 2015

M.Sc., McMaster University, 2018

Submitted to the Graduate Faculty of

the Dietrich School of Arts and Sciences in partial fulfillment

of the requirements for the degree of

Doctor of Philosophy

University of Pittsburgh

2023

UNIVERSITY OF PITTSBURGH

DEPARTMENT OF MATHEMATICS

This dissertation was presented

by

Jesse Michael Han

It was defended on

April 7, 2023

and approved by

Thomas Hales, Mellon Professor Mathematics

Kiumars Kaveh, Professor of Mathematics

Bogdan Ion, Associate Professor of Mathematics

Jeremy Avigad, Professor of Philosophy (Carnegie Mellon University)

Contributions to Neural Theorem Proving

Jesse Michael Han, PhD

University of Pittsburgh, 2023

In this dissertation we present several contributions to the nascent field of neural theorem proving, *i.e.* deep learning-driven automated theorem proving over large libraries of formalized mathematics. We work primarily with the Lean theorem prover along with its standard mathematical library, mathlib. We focus on addressing the issue of *data scarcity* for training autoregressive language models to perform high-level reasoning steps at the same level of abstraction as humans. Labeled data for imitation learning of human proof steps is extremely scarce as formal mathematics libraries require years of concentrated effort by human specialists to be built. We show that this paucity of data can be addressed by (1) extracting data from mathlib's low-level proof artifacts, (2) synthesizing those low-level proof artifacts from a language model, and (3) creating new training data from autonomous proof attempts in an expert iteration loop. We demonstrate an application of these kinds of techniques to achieve state-of-the-art performance on solving formalized Olympiad-level mathematics problems.

Table of Contents

Preface . . . xiii

1.0 Introduction . . . 1

2.0 Background . . . 3

2.1 Formal theorem proving . . . 3

2.1.1 Systems for formal theorem proving . . . 3

2.2 Deep learning and generative language modeling . . . 5

2.2.1 Autoregressive sequence modeling . . . 5

2.2.2 Transformer language models . . . 6

2.3 Prior and related work . . . 8

2.4 Conclusion . . . 10

3.0 Data augmentation using proof artifacts . . . 11

3.1 Proof artifact co-training for theorem proving with language models . . . 11

3.1.1 Introduction . . . 11

3.1.2 Background and related work . . . 12

3.1.2.1 The LeanStep datasets and machine learning environment 15

3.1.2.2 Machine learning environment . . . 16

3.1.3 Experiments . . . 18

3.1.4 Discussion . . . 22

3.1.5 Acknowledgments . . . 24

3.2 Synthetic proof term data augmentation . . . 24

3.2.1 Introduction and background . . . 24

3.2.1.1 Comparison to related work . . . 25

3.2.2 Methodology . . . 26

3.2.2.1 Bootstrap Datasets . . . 26

3.2.2.2 Creating Synthetic Datasets . . . 27

3.2.3 Experimental Setup . . . 28

3.2.3.1 Splitting the Bootstrap Data . . . 28
3.2.3.2 Proof Term Language Modelling Experiments . . . 29
4.0 Statement curriculum learning and applications to Olympiad theorem proving . . . 36
4.0.1 Introduction . . . 36
4.0.1.1 *miniF2F* benchmark . . . 38
4.0.1.2 Contribution . . . 38
4.0.2 Related Work . . . 39
4.0.3 Formal Environment . . . 40
4.0.3.1 `lean-gym` . . . 40
4.0.3.2 Proof datasets extraction . . . 42
4.0.4 Expert Iteration . . . 42
4.0.4.1 Model . . . 42
4.0.4.2 Pre-Training . . . 42
4.0.4.3 Training objectives . . . 43
4.0.4.4 Bootstrapping . . . 45
4.0.4.5 Iterated sampling and training . . . 47
4.0.4.6 Expert iteration on *mathlib-train* . . . 48
4.0.5 Statement curriculum learning . . . 51
4.0.5.1 Synthetic inequality generator . . . 51
4.0.5.2 Expert iteration on synthetic inequality statements . . . 52
4.0.6 Targeting *miniF2F* . . . 53
4.0.6.1 Formalization effort . . . 53
4.0.6.2 Transfer to *miniF2F* . . . 54
4.0.6.3 Results . . . 56
4.0.7 Discussion . . . 56
4.0.7.1 Model Size . . . 56
4.0.7.2 Qualitative analysis of proofs . . . 58
4.0.7.3 Limitations . . . 59
5.0 Conclusions . . . 61

A.1 Additional materials for section 3.1 . . . 63
A.1.1 Additional background for PACT . . . 63
A.2 Datasets for section 3.1 . . . 65
A.2.1 Pre-training datasets . . . 65
A.2.2 Dataset sizes . . . 66
A.2.3 Example datapoints . . . 67
A.3 Experiments for section 3.1 . . . 71
A.3.1 Chained tactic prediction . . . 71
A.3.2 Theorem naming case study . . . 73
A.3.3 Test set evaluation breakdown by module . . . 76
A.3.4 Baseline description . . . 76
A.3.5 Computational resource estimates . . . 78
A.4 Example proofs for section 3.1 . . . 79
A.4.1 `lie_algebra.morphism.map_bot_iff` . . . 79
A.4.2 `primrec.of_equiv` . . . 79
A.4.3 `real.tan_eq_sin_div_cos` . . . 80
A.4.4 `sym2.is_diag_iff_proj_eq` . . . 81
A.4.5 `norm_le_zero_iff` . . . 81
A.5 Additional materials for chapter 4 . . . 82
A.5.1 Lean-gym . . . 82
A.5.2 Synthetic inequalities . . . 83
A.5.2.1 List of inequality composition theorems . . . 83
A.5.2.2 Examples of synthetic inequalities . . . 84
A.5.3 $N_D = 0$ $N_S = 0$ . . . 84
A.5.4 $N_D = 0$ $N_S = 4$ . . . 85
A.5.5 $N_D = 4$ $N_S = 4$ . . . 86
A.5.6 Example proofs from *mathlib-train* . . . 87
A.5.7 `comap_eq_of_inverse` . . . 88
A.5.8 `sum_range_sub_sum_range` . . . 89
A.5.9 `prod_inv_distrib` . . . 90

A.5.10 Example proofs from *minif2f-{test,valid,curriculum}* 91
A.5.10.1 `imo_1961_p1` 91
A.5.10.2 `imo_1964_p2` 93
A.5.10.3 `aime_1990_p15` 94
A.5.10.4 `mathd_train_algebra_217` 96
A.5.10.5 `amc12b_2020_p6` 97
A.5.10.6 `mathd_algebra_140` 98
A.5.10.7 `aime_1984_p1` 100
A.5.10.8 `aopsbook_v2_c8_ex1` 101
A.5.10.9 `mathd_numbertheory_447` 103
Bibliography 104

List of Tables

1 Test loss, test perplexity, and test accuracy (if appropriate) of the best model checkpoint trained on each dataset. 35

2 Test loss, test perplexity, and test accuracy of the best model checkpoint trained on each dataset with different levels of dropout. "Bootstrap X" rows indicate test performance after training on cond_train_bootstrap. "Augmented X" rows indicate test performance after training on cond_train_aug. "X Gap" rows are computed by taking the absolute difference between "Augmented X" and "Bootstrap X". 35

3 Mix and source of data involved in the updated *WebMath* pre-training. 43

4 Performance of θ_0 and θ_1 on *mathlib-valid* and *miniF2F-valid* compared to PACT Lean GPT-f as reported in [1, 2]. All models have the same architecture. θ_0 is sampled using cumulative logprob priority best-first search. θ_1 is sampled using best-first search based on the *proofsize objective*. We report our setup ($d = 512$ expansions and $e = 8$ tactic samples per expansions) as well as the setups used in [1, 2] to control for compute. We also report the performance of θ_1 on *mathlib-valid* when trained using the *outcome objective* from [3] as an ablation of our proposed *proofsize objective*. 46

5 Performance of θ_1 (value-function based search), $\theta_9^{mathlib}$ (expert iterated on *mathlib-train*) and θ_9^{full} (expert iterated on our full curriculum) on *mathlib-{valid, test}* and *miniF2F-{valid, test}*. All proof searches are run with $d = 512$ and $e = 8$. 57

6 Counting the number of semicolon-chained tactics predicted by our models that appear *in successful proofs*. Each column headed by a number n; indicates the number of times that a suggestion appeared with n occurrences of ';'. 72

List of Figures

1 Auto-regressive objectives used for each task described in subsubsection 3.1.2.1. Placeholders represented with brackets (such as <TacticState>) are substituted by the context-completion pairs from each datasets in the prompts above. Each task is presented to the model with its respective keyword (PROOFSTEP, NEXTLEMMA,...). We wrap the completions of mix1 tasks (with apply(...) and exact(...) respectively) as a hint that they are related to the respective Lean tactics; this is not directly possible for the other tasks. 20

2 Comparison of pre-training and co-training on mix-1 and mix-2. We use > to separate distinct pre-training phases and + associatively indicates that a co-training mixture. As an example, WebMath > mix2 > mix1 + tactic signifies a model successively pre-trained on WebMath then mix2 and finally co-trained as a fine-tuning step on mix1 and tactic. Columns mix1, mix2, tactic report the optimal validation loss achieved on these respective datasets. We provide a detailed description of experiment runtime and computing infrastructure in section A.2. "Tokens elapsed" means the number of tokens the model was trained on. 31

3 Validation losses achieved in the pre-training and co-training setups without WebMath pre-training. See Figure 2 for a description of the columns and the models nomenclature used. †Due to technical constraints, we are unable to provide pass-rates for some of the models. We use > to separate distinct pre-training phases and + associatively indicates that a co-training mixture. "Tokens elapsed" means the number of tokens the model was trained on. 32

4 Validation losses and pass-rates achieved for various model sizes using PACT. See Figure 2 for a description of the columns. The setup used is `WebMath > mix1 + mix2 + tactic`. We use `>` to separate distinct pre-training phases and `+` associatively indicates that a co-training mixture. "Tokens elapsed" means the number of tokens the model was trained on. 33

5 As sampling temperature is increased, both typecheck pass rate and similarity of each type-correct generated example to the most similar training example (i.e. "max cosine similarity") decreases. Note that the mean "max cosine similarity" is the average observed max cosine similarity across the type-correct generated examples with a given temperature. 34

6 *pass@1* (plain) and *pass@8* (dotted) for *mathlib-valid* and *minif2f-valid* when running eight expert iterations with *St* set to be the statements in *mathlib-train*. The x-axis is log-scaled. It corresponds to the indices of the θ_k models and serves as a good proxy to compute (the amount of test-time and train-time compute per iteration being fixed). The y-axis is scaled linearly and simply shifted between the two graphs (spans an equal range). 49

7 Cumulative pass rate for our *expert iteration* loop as well as a *sample only* loop where we skip re-training the model between iterations. The *adjusted compute* line is computed by fitting the *sample only* curve and shifting it to approximate a setup where we would focus all the additional compute used by expert iteration (sampling training data from *mathlib-train* as well as re-training models at each iteration) towards running proof searches against *mathlib-valid*. 49

8 Cumulative pass rate for our *expert iteration* loop as well as a *sample only* loop where we skip re-training the model between iterations. Pass rates are reported for each value of N_D (pooling together $0 \leq N_S \leq 7$). 53

9 **Top:** cumulative pass rate on *miniF2F-valid* for our expert iteration loop using our full curriculum (*mathlib-train*, *synth-ineq* and *miniF2F-curriculum*) compared to the expert iteration loop from section subsubsection 4.0.4.6. The total number of attempts per iteration in our *full* loop is $25k + 5.6k + 8 * 327 \approx 33.2k$, which means the total compute deployed is higher than in the *mathlib-train* only loop ($25k$). We therefore also report in dotted a *mathlib-train* only loop, taking $a = 2$, whose total number of attempts per iteration is $\approx 50k$. **Bottom:** *pass@1* (plain) and *pass@8* (dotted) for our expert iteration loop using our full curriculum (*mathlib-train*, *synth-ineq* and *miniF2F-curriculum*) compared to the *expert iteration* loop from section subsubsection 4.0.4.6. 55

10 A sample of correct top-1 guesses by our best model `wm-to-tt-m1-m2` on the *theorem naming* task. We performed this experiment on the `future-mathlib` evaluation set, which comprises entirely unseen theorems added to `mathlib` only after we last extracted training data. 74

11 A sample of incorrect guesses by our best model `wm-to-tt-m1-m2` on the *theorem naming* task. We performed this experiment on the `future-mathlib` evaluation set, which comprises entirely unseen theorems added to `mathlib` only after we last extracted training data. Most of the top-8 guesses displayed in the above table are very similar to the ground truth, in some cases being equivalent up to permutation of underscore-separated tokens. Note that for the first example, the concept of `ordnode` was not in the training data whatsoever and all predictions are in the syntactically similar `ordinal` namespace. 75

12 A breakdown of theorem proving success rate on the `test` set for `wm-to-tt-m1-m2`, `wm-to-tt-m1`, `wm-to-tt`, and the `tidy` baseline across top-level modules in Lean's `mathlib`. We see that `wm-to-tt-m1-m2` mostly dominates `wm-to-tt-m1` and the models trained using PACT dominate the model `wm-to-tt` trained on human tactic proof steps. 77

Preface

I started this journey with the dream that one day human mathematicians would work alongside artificially intelligent ones. I saw a future where machines would search for proofs at a planetary scale, eventually going beyond proof search to synthesize compelling definitions and abstractions in a self-reinforcing loop, and where one day mathematics would become a shared, formally verified, and mostly autonomous computational construct, like the operating system of a mainframe computer whose details are hidden from the end users. This dream is not as far-fetched as it may seem at first. After all, machines already perform many tasks that were once done by human beings, and they are only getting closer to the remainder. This dream is still well within the realm of possibility, to which the contributions presented in this dissertation hopefully bring us a few steps closer.

Along the way, I have benefitted and grown tremendously from the guidance and mentorship of many remarkable individuals. Jeremy Avigad was always warm and encouraging and played a pivotal role in organizing the formal verification research community in Pittsburgh. I am grateful to Floris van Doorn and Mario Carneiro, who taught me the ins and outs of theorem proving in Lean and how to think like a proof engineer. Floris was an invaluable collaborator as we formalized the independence of CH in Lean. Simon Hudon, who remains one of the most talented engineers I have ever met, was always generous with his time, energy, and ideas, and taught me the value of curiosity and rapid prototyping for research. I am similarly grateful to Reid Barton for the occasional late-night conversations that gave me the sense of watching a mathematician fifty years from the future at play.

I am also grateful to the mentors who took a chance on me as I found my way to the bleeding edge of research in artificial intelligence and automated reasoning. John Harrison generously advised my research on applying neural networks to SAT solvers during the months I spent with the AWS Automated Reasoning group. Dan Selsam taught me the craft of automated theorem proving over the course of a summer-long apprenticeship spent building ARC and IMO geometry solvers in Haskell, and led by example with lion-hearted tenacity. Christian Szegedy, Markus Rabe, and Tony Wu taught me how to think like an AI researcher

and were invaluable advisors as I explored neural theorem proving in Lean. Stan Polu taught me the value of speed, execution, and focus as we built out the formal math research program at OpenAI. Igor Babuschkin taught me the value of simplicity and being restlessly ambitious in engineering and research. Ilya Sutskever taught me the value of conviction in an idea compounded over time.

Finally, I would not be the researcher or person I am today without the support of my advisor, Tom Hales. His persistence, clarity of thought, and insistence on excellence have long served as a model for mine. Thank you, Tom—for giving me license to set and pursue my own research agenda, for unwaveringly supporting my work no matter how audacious the timelines or goals, and for supplying the courage for a lifetime of impossibly ambitious tasks.

1.0 Introduction

Neural theorem proving is a newly developing field at the intersection of machine learning and automated theorem proving. The goal of neural theorem proving is to endow theorem provers with powerful deep learning models that can automatically generate human-level proofs for formalized mathematical theorems.

Theoretical advances in neural theorem proving are being driven by a number of exciting recent developments in machine learning. On the one hand, there has been a dramatic increase in the performance of neural networks for natural language processing tasks such as machine translation, text classification, and question answering. On the other hand, there has been a resurgence of interest in neural networks for automated theorem proving, driven in part by the success of deep learning models in other areas of AI.

The goal of this dissertation is to present several contributions to the nascent field of neural theorem proving. We work primarily with the Lean theorem prover along with its standard mathematical library, mathlib. We focus on addressing the issue of *data scarcity* for training autoregressive language models to perform high-level reasoning steps at the same level of abstraction as human proof engineers. The quality of deep learning models (for a fixed architecture) is a function of compute and training data quality — ImageNet, for example, has over a million images in its training set for the far less reasoning-intensive task of image classification. We show that the paucity of human-written data in formal theorem proving can be addressed in three ways.

1. Extracting data from mathlib's low-level proof artifacts. As part of the proof-checking process in Lean, the higher-level tactic commands used by a human formalized are compiled to a far more detailed proof artifact which is verified by Lean's kernel. These low-level proof artifacts can be analyzed in various ways to extract alternate proofs and reformulations of the original proof which provide a rich source of related training data. We show that adding these kinds of data can have a massive impact on theorem proving performance of a neural theorem prover powered by a generative language model.
2. Synthesizing low-level proof artifacts from a language model. While one can obtain a very

large number of datapoints via the above method, it is still fundamentally constrained by the number of high-level proofs supplied by a human. We show that one can simply train a generative language model on the distribution of all low-level proof terms extracted from mathlib and sample new syntactically correct and kernel-verified low-level proof terms from this model and that they provide training data which is helpful for a generative language model on theorem proving tasks.

3. Creating new training data from autonomous proof attempts in an expert iteration loop. We introduce an expert iteration methodology called *statement curriculum learning* which bootstraps a neural theorem prover by iteratively running the latest version of that prover against a stationary distribution of theorem statements and using any newly discovered proofs as training data to train the next version of the prover. We show that interleaving proof search with model training proves significantly more theorems than a stationary model performing proof search alone.

Finally, we demonstrate an application of these kinds of techniques to achieve state-of-the-art performance on a benchmark of formally stated theorem statements derived from math Olympiad competitions.

2.0 Background

In this chapter we review the necessary background for the rest of the thesis.

2.1 Formal theorem proving

Formal theorem proving refers to the practice of programming in a computer language which allows for computer-readable specifications of mathematical theorems and computer-verifiable representations of proofs [4].

2.1.1 Systems for formal theorem proving

There are various software packages implementing formal verification systems that can be applied to the verification of mathematical proofs. We discuss several prominent ones that have served as the starting point for various neural theorem provers. In this dissertation we focus on the Lean 3 theorem prover, which implements a dependent type theory.

- The **Lean theorem prover** is a functional programming language that allows for easy writing of correct and maintainable code [5]. It can also be used as an interactive theorem prover. Some features of Lean include type inference, first-class functions, powerful data types, pattern matching, type classes, extensible syntax, hygienic macros, dependent types, metaprogramming framework, multithreading, and verification [6]. The Lean project was launched in 2013 by Leonardo de Moura at Microsoft Research and is hosted on GitHub. The latest version, Lean 4, was released on January 4, 2021. We emphasize that this dissertation focuses on neural theorem proving in the Lean 3 theorem prover [6]. Lean's foundations comprise a *dependent type theory* along with the calculus of inductive constructions, a countable hierarchy of non-cumulative universes, and quotient types [7]. For an elementary introduction to dependent type theory, we refer the user to the textbook by Barthe and Coquand [8]. We also refer to [7] which gives a rigorous syntactic definition

of Lean's precise type theory and which demonstrates an equiconsistency result between Lean's type theory and ZFC extended with a hierarchy of countably many Grothendieck universes.

A few other formal theorem proving systems (not at all inclusive) include:

- **HOL Light** is a proof assistant and computer program designed to help users prove theorems in higher order logic [9]. It is based on a formulation of simple type theory with equality as the only primitive constant, and uses the OCaml programming language. It is distinguished by its clean and simple design and small logical kernel. Despite this, it provides powerful proof tools and has been applied to formalizing mathematics and industrial verification [10, 11].
- **Coq** is a formal proof management system with a language to write mathematical definitions, executable algorithms, and theorems, as well as an environment for developing machine-checked proofs. It is free and open-source, with development largely supported by INRIA, a French national research institute. It has also been applied to formalizing mathematics and industrial verification [12, 13].
- **Metamath** is a formal language used for archiving, verifying, and studying mathematical proofs. It is based on set theory and first-order logic, and has been used to formalize more than 23,000 proofs from mathematics. Metamath has been used to develop databases of proved theorems in fields such as logic, set theory, number theory, algebra, topology, and analysis [14].

For further background, we refer the reader to surveys on formal verification in mathematics [4, 15].

To give more intuition about theorem proving in Lean, we emphasize that any type theory, Lean included, is built around a collection of (well-formed) *expressions*. These are strings of symbols which have been produced by a finite sequence of expression formation rules, and are therefore trees of subexpressions. All expressions have a *type*, which is another expression. Lean is equipped with a countable non-cumulative hierarchy of (Grothendieck) universes, with a type `Prop` of propositions at the bottom. Mathematical assertions like theorems are translated into Lean by specifying a type of type `Prop`, and to supply a proof

of that theorem is to supply an expression whose type is that proposition. Lean is *proof-irrelevant*, meaning that any two proofs of a given proposition (i.e. expressions belonging to the same propositional type) are definitionally equivalent. Many provers, such as Lean, support a mode of working where the theorem to be proved is presented as a "goal" that is transformed by applying "tactics" in a backward fashion [16]. From the perspective of expression synthesis, this corresponds to transforming a "metavariable" with the original type through the application of "tactic" metaprograms, producing sub-metavariables, and repeating this process until there are no metavariables remaining. We refer the reader to the thesis of Ayers for a more detailed exposition on proof assistants and how partial proofs are handled with metavariables in Lean [17].

2.2 Deep learning and generative language modeling

2.2.1 Autoregressive sequence modeling

Autoregressive sequence modeling is a type of machine learning that is used to predict the next value in a sequence. This is based on the assumption that future values in a sequence can be predicted based on the past. In autoregressive sequence modeling, the output of the model at time t is a function of the input at time t and the outputs at all previous time steps. This makes it well suited for tasks such as language modeling, where the next word in a sentence is predicted based on the previous words.

Formally, given a probability distribution $\mathfrak{D}$ on X^*, the set of all finite sequences on a set X, we define an autoregressive sequence model $\mathfrak{M}$ of $\mathfrak{D}$ to be a function $\mathfrak{M} : X^* \to (X \to [0,1])$ which assigns a probability distribution over X to any finite sequence $(x_1, \ldots, x_N)$ which is an approximation of the true conditional probability distribution $\mathfrak{D}(x_{N+1}|x_1, \ldots, x_N)$. Using $\mathfrak{M}$, we can define a joint probability distribution over all finite sequences using the chain rule of conditional probability:

$$P_{\mathfrak{M}}(x_0, \ldots, x_N) = \mathfrak{M}(\emptyset)(x_0) \cdot \mathfrak{M}((x_0))(x_1) \cdots \mathfrak{M}((x_0, \ldots, x_{N-1}))(x_N)$$

which approximates $\mathfrak{D}$ over X^*.

For example, X could be set of all Unicode characters so that X^* is the set of all strings of Unicode characters, and $\mathfrak{D}$ could be a probability distribution supported only on strings which have appeared on the Internet or published in a paper on the arXiv. When the target $\mathfrak{D}$ is derived from language, we call any model $\mathfrak{M}$ thus defined an *autoregressive language model*. Of particular interest to us are distributions $\mathfrak{D}$ supported on strings which represent statements and proofs of theorems in a formal theorem proving system like Lean, and autoregressive sequence models thereof. In this thesis, we focus especially on modeling the distribution $\mathfrak{D}_{\text{tactic}}$ of strings, defined to be the set of all strings of the form $p :: s$ (a prefix p concatenated with a suffix s), with a prefix which is a string representation of a tactic state from the Lean theorem prover and a suffix which is a string representation of the next tactic application. For more on tactics in Lean, we refer the reader to subsection 3.1.2.

In this dissertation we focus on the problem of approximating $\mathfrak{D}_{\text{tactic}}$ using *causal decoder-only transformer language models*. We now turn to discussing transformer language models, which currently underlie the state of the art in autoregressive sequence modeling.

2.2.2 Transformer language models

In a nutshell, transformers are deep learning models that use the mechanism of self-attention. We will define these terms momentarily. They are used primarily in natural language processing and computer vision. Transformers were introduced in 2017 and have become the model of choice for many natural language processing (NLP) problems. The transformer architecture was first proposed in the paper "Attention Is All You Need" [18]. It is designed to solve sequence-to-sequence tasks while handling long-range dependencies with ease and is now the basis of many state-of-the-art methods in natural language processing.

In this dissertation, we focus on (causal) *decoder-only* transformers (which we refer to simply as transformers). These are precisely autoregressive sequence models for a fixed length. Formally, given a sequence length L, a hidden dimension D, a depth N, a transformer is a function T_θ from sequences of length N over a finite *set of tokens* X to a sequence of length N of probability distributions over X. The transformer T_θ is parametrized by a set parameters θ which we learn through stochastic gradient descent [19, 20] (SGD) on training data. It

operates roughly as follows. Let $(x_1, \ldots, x_L)$ be the input sequence. We apply a learnable function $e : X \to \mathbb{R}^D$ to represent each sequence element as an *embedding* in $\mathbb{R}^D$. We also add a learnable positional encoding $[pos_1, \ldots, pos_L] \in \mathbb{R}^{L \times D}$ to $[e(x_1), \ldots, e(x_L)]$.

We then transform the embeddings $[e(x_1), \ldots, e(x_L)]$ by applying N different learnable *transformer blocks* $\left(B_i : i \in \{1, \ldots, N\} : \mathbb{R}^{L \times D} \to \mathbb{R}^{L \times D}\right)$ to $[e(x_1), \ldots, e(x_L)]$ in sequence, producing a final set of embeddings

$$[\widehat{x}_1, \ldots, \widehat{x}_n] := B_N(B_{N-1}(\ldots B_1(E))), \text{ where } E := [e(x_1), \ldots, e(x_n)].$$

We apply a learnable feedforward network which is just a function $\mathsf{unembed} : \mathbb{R}^D \to \mathbb{R}^{|X|}$ row-wise to $[\widehat{x}_1, \ldots, \widehat{x}_n]$ to produce an array in $\mathbb{R}^{L \times |X|}$. Each row X_k of this array defines a function $X \to \mathbb{R}$ which can be converted to a probability distribution P_k using the softmax function

$$\text{softmax}(x)_i = \frac{e^{x_i}}{\sum_j e^{x_j}}$$

and each probability P_k is precisely the conditional probability distribution over tokens X given the previous elements $x_1, \ldots, x_{k-1}$ of the sequence.

Given a corpus $\vec{X}$ of training data, we train the parameters of a transformer via SGD to minimize the categorical cross-entropy loss[1] between the conditional probability P_k emitted by T at position k and the categorical distribution for the true label $\vec{X}_k$ (i.e. the discrete probability distribution with all mass concentrated at the kth position) conditioned on $\vec{X}_1, \ldots, \vec{X}_{k-1}$, simultaneously at all positions k.

We refer the reader to [18] for more detail on transformer blocks and the applications of the transformer architecture to natural language processing.

Generative language models build with transformer architectures have recently been the focus of much work deep learning and natural language processing, in light of remarkable emergent abilities (e.g. multi-task learning [21, 22], in-context metalearning [23], and predictable scaling with data and compute [24]) when training at large scale (e.g. with hidden dimension D in the thousands dozens of layers N, billions of trainable parameters and hundreds of

[1]The cross entropy loss between two discrete probability distributions P and Q on a finite set X is defined as: $L(P, Q) = -\sum_{x \in X} P(x) \log Q(x)$. The categorical cross entropy loss is the special case where P is zero for all but one $x \in X$, so minimizing the categorical cross entropy is equivalent to maximizing the log-likelihood of the true label.

billions to trillions worth of tokens of training data scraped from the internet [23]). In this dissertation we focus on taking such large pre-trained models and adapting them to modeling the distribution $\mathfrak{D}_{\text{tactic}}$ of strings representing proof-states and proof-steps from the Lean theorem prover.

2.3 Prior and related work

The work in [25] demonstrated the utility of language models for learning to perform tasks requiring mathematical reasoning. They train and evaluate language models on a collection of different tasks but do not directly attempt transfer to the task of theorem proving. The GPT-f (*i.e.* **g**enerative **p**re-**t**raining for **f**ormal math) work [26] was the first result to directly demonstrate the utility of language models for theorem proving. The language modelling task for GPT-f was based around reproducing individual proof steps in Metamath proofs. After training such a model, it could be used to perform a tree-based proof search. Generated proof steps which were invalid could be filtered out with the use of a formal verifier, similar to our use of the Lean kernel to filter proofs that fail typechecking.

In [27, 28], first-order theorem provers are trained from scratch with procedurally generated theorems. The forward proposer which generates theorems samples randomly from the available axioms at each step and does not leverage a learned model. INT [29] procedurally generates synthetic inequality-based theorems to systematically investigate out-of-distribution generalization. LIME [30] procedurally generates examples for three classes of tasks (deduction, abduction, and induction) and uses them to pre-train transformer models before finetuning on other tasks. The authors demonstrate this is a useful replacement for more expensive natural language pre-training tasks, however it does not enable the production of formal mathematical data.

Another category of work focuses on synthesizing conjectures [31, 32]. However, the existence of proofs for the generated conjectures cannot be guaranteed and must be found via search. Also, several works re-train models using alternative proofs found by trained models [26, 33, 34, 35]. However this doesn't enable discovery of new theorems, only variant

proofs of existing theorems, hence cannot fundamentally alleviate problem of data scarcity in automated theorem proving.

Prior work by Wang and Deng [36] was the first to demonstrate the utility of a neural generator of synthetic training data for theorem proving. The generator is limited by the requirement that new hypotheses are produced by explicit substitution and in order to avoid generating "meaningless theorems" new hypotheses are required be identical to part of an existing proof tree.

Other machine learning and neural theorem provers for tactic-based interactive theorem provers take one of two approaches to tactic generation. TacticToe [37] for HOL4 and Tactician [38] for Coq use k-NN to select similar tactics in the training set and apply modifications to the result, *e.g.* swapping the tactic variables with those found in the local context. HOList/DeepHOL [39, 40, 41] for HOL Light; TacticZero [42] for HOL4; and CoqGym/ASTactic [43] and ProverBot9001 [44] for Coq hard-code the DSL (*i.e.* domain-specific language) for every tactic command. The model chooses a tactic command, and then fills in the tactic arguments using specialized argument selectors (such as a lemma selector, a local hypothesis selector, and/or a variable selector). None of these selectors currently synthesize arbitrary terms, which prevents the overall tactic synthesis procedure from constructing tactics with proof terms, such as `@norm_eq_zero` α `_ g`, or directly proving an existential, *e.g.* $\exists$ `(x : ℝ), x + 3 = 0`, by supplying the witnessing term `-3`.

Language models have also been explored in the first-order ITP (*i.e.* interactive theorem prover) Mizar for conjecturing and proof synthesis [45]. While their work shows the promise of such approaches, is not intended as a complete end-to-end theorem prover. For Metamath, which does not use tactics, language modeling approaches have been quite successful. Holophrasm [46], MetaGen [47], and GPT-f [48] all use RNNs (*i.e.* recurrent neural networks) or transformers to generate proof steps. Indeed, we build on the work of Metamath GPT-f [48] (MM GPT-f).

2.4 Conclusion

We can frame the central problem of neural theorem proving with autoregressive language models as follows. How do we (1) use recent advances in generative language modeling to induce powerful proposal distributions over proof-steps conditioned on proof-states, and (2) design algorithms that can apply such distributions to human-level proof synthesis?

We will spend the remainder of this dissertation addressing both points. We will present various data augmentation methods to produce better proposal distributions over proof-steps using large generative language model, an implementation of a neural best-first search for theorem proving in Lean itself, and demonstrate a concrete application to state-of-the-art theorem proving performance on a benchmark of Olympiad-level mathematics problems.

3.0 Data augmentation using proof artifacts

3.1 Proof artifact co-training for theorem proving with language models

Labeled data for imitation learning of theorem proving in large libraries of formalized mathematics is scarce, as such libraries require years of concentrated effort by human specialists to be built. This is particularly challenging when applying large transformer language models to tactic prediction, because the scaling of performance with respect to model size is quickly disrupted in the data-scarce, easily-overfitted regime. We propose PACT (**P**roof **A**rtifact **C**o-**T**raining), a general methodology for extracting abundant self-supervised data from kernel-level proof terms for joint training alongside the usual tactic prediction objective. We apply this methodology to Lean, a proof assistant host to some of the most sophisticated formalized mathematics to date. We instrument Lean with a neural theorem prover driven by a transformer language model and show that PACT improves theorem proving success rate on a held-out suite of test theorems from 32% to 48%.

Some material in section appears in the paper [49], published at ICLR 2022, and this work is joint with Jason Rute, Yuhuai Wu, Edward Ayers, and Stanislas Polu.

3.1.1 Introduction

Deep learning-driven automated theorem proving in large libraries of formalized mathematics (henceforth "neural theorem proving") has been the focus of increased attention in recent years. Labeled data for imitation learning of theorem proving is scarce—formalization is notoriously labor-intensive, with an estimated cost of 2.5 man-years per megabyte of formalized mathematics [50], and complex projects require years of labor from human specialists. Within a fixed corpus of (possibly unproven) theorem statements, it is possible to augment a seed dataset of human proofs with new successful trajectories using reinforcement learning or expert iteration. However, for some large models this can be quite computationally intensive, and without a way to expand the curriculum of theorems, the agent will inevitably saturate

and suffer from data starvation.

Data scarcity is a particularly thorny obstruction for applying large language models (LLMs) to neural theorem proving. LLMs have achieved spectacular success in data-rich regimes such as plain text [23], images [51], and joint text-image modeling [52]. The performance of decoder-only transformers has been empirically shown to obey scaling power laws in model and data size [53]. However, existing datasets of human proof steps for neural theorem proving are extremely small and exist at scales at which overfitting occurs extremely rapidly, disrupting the scaling of performance with respect to model size [54].

We make two contributions towards addressing the problem of data scarcity in the context of formal mathematics. First, we introduce PACT, a general methodology for extracting self-supervised auxiliary tasks for jointly training a language model alongside a tactic prediction objective for interactive theorem proving. Second, we present LeanStep, a collection of datasets and a machine learning environment for the Lean 3 theorem prover with support for PACT, supervised learning of tactic prediction, theorem proving evaluation, and reinforcement learning.

We train large language models on these data and demonstrate that PACT significantly improves theorem proving success rate. We then embark on a careful study of the effects of pre-training vs. co-training and show that PACT combined with *WebMath* pre-training [48] achieves the best validation loss and theorem proving success rate. Finally, on an out-of-distribution collection of thousands of theorems (some involving novel definitions) added to Lean's mathematical library after we extracted our train/test data, we achieve a theorem proving success rate of 37%, suggesting strong generalization and usefulness at the frontier of formalized mathematics.

3.1.2 Background and related work

Lean is an interactive theorem prover and functional programming language [55]. It has an extremely active community and is host to some of the most sophisticated formalized mathematics in the world, including scheme theory [56], forcing [57], perfectoid spaces [58], and condensed mathematics [59]. Lean's foundational logic is a dependent type theory called

the calculus of inductive constructions [60]. This design means that terms, types and proofs are all represented with a single datatype called an *expression*. A *proof term* is a Lean expression whose type is a proposition, *i.e.* a theorem. This proof term serves as a checkable artifact for verifying the proposition. Lean uses a small, trusted kernel to verify proof terms. The primary repository of formalized mathematics in Lean is `mathlib` [61]. At the time of writing, 140 contributors have added almost 500,000 lines of code; `mathlib` contains over 46,000 formalized lemmas backed by over 21,000 definitions, covering topics such as algebraic geometry, computability, measure theory, and category theory. The range of topics and the monolithic, unified organization of `mathlib` make it an excellent foundation for a neural theorem proving dataset.

In a tactic-based interactive theorem prover (ITP) such as Lean, a proof is a list of tactics, *i.e.* small proof-term-generating programs. Tactics can be simple one-word commands, *e.g.* `refl` (the tactic which invokes a check for definitional equality), or be composed of many nested parts, *e.g.*

```
simpa [le_antisymm_iff, norm_nonneg] using @norm_eq_zero α _ g
```

Here the brackets enclose a list of simplifier rules (which often are just lemmas from the library), and `@norm_eq_zero` α `_ g` is a proof term applying the lemma `norm_eq_zero` to the local variables α and `g`.

We remark that our tactic generator is able to synthesize tactics of any form found in `mathlib` including, for example, the `simpa` example above as a one line proof to a test theorem, even though the string `@norm_eq_zero` does not occur in our dataset. (See more examples in section A.4.) We leave as future work the possibility of re-integrating specialized components, *e.g.* lemma selection, found in other works (possibly as, say, a source of additional prompts for the language model). Directly applying generative language modeling to tactic generation allows this setup to be considerably simplified.

Whereas MM GPT-f [48] trained primarily on the Metamath proof step objective (*i.e.* guessing the next lemma to be applied to a goal, which is similar to our `NEXTLEMMA` task in subsubsection 3.1.2.1), we co-train on a diverse suite of self-supervised tasks extracted from Lean proof terms and demonstrate significant improvements in theorem proving performance

when doing so. This is our main result.

Besides theorem proving, a number of recent papers have shown that language models, especially transformers, are capable of something like mathematical and logical reasoning in integration [62], differential equations [63], Boolean satisfiability [64], and inferring missing proof steps [65].

A closely-related vein of work has shown that pre-training transformers on data engineered to reflect inductive biases conducive to mathematical reasoning is beneficial for downstream mathematical reasoning tasks [66, 67]. Our work both builds on and departs from these ideas in several ways. Unlike skip-tree training (c.f. [66]), which focuses solely on predicting masked subterms of theorem *statements*, PACT derives its self-supervised training data from far more complex *proofs*. Unlike LIME [67], which uses purely synthetic data and is presented as a pre-training methodology, our self-supervised tasks are extracted from non-synthetic human proofs. Moreover, we show that not only are transformers capable of performing well on auxiliary tasks gathered from low-level proof artifact data, but that we can directly leverage this low-level data by jointly training a language model to greatly improve its performance at high-level theorem proving.

The idea of mining low-level proof artifacts was previously explored by Kaliszyk and Urban in the context of automated lemma extraction [68, 69]. It has also been previously observed that training on fully elaborated Coq terms [70] helps with a downstream theorem naming task. However, similar to previous work on skip-tree training, their dataset focuses solely on theorem statements, *i.e.* types, does not cover the far more complex proof terms, and does not evaluate the effect of such training on theorem proving evaluations.

While there exist environments and datasets for other formal mathematics libraries [71, 65, 72, 73], LeanStep is the first and only tactic proof dataset for the Lean theorem prover. This makes available a large set of formal mathematical data to researchers covering a diverse and deep spectrum of pure mathematics. Moreover, LeanStep is unique in that it contains both high-level human-written tactics as well as kernel-level proof terms, which enables the extraction of self-supervised tasks for PACT (subsubsection 3.1.2.1).

3.1.2.1 The LeanStep datasets and machine learning environment

Tactics in Lean are metaprograms [6], which can construct Lean expressions, such as proof terms. A *tactic state* which tracks the list of open goals and other metadata (like the partial proof term constructed so far) is threaded through each tactic invocation. Lean has special support for treating tactics as an extensible domain-specific language (DSL); this DSL is how Lean is typically used as an interactive theorem prover. The DSL amounts to a linear chain of comma-separated invocations. The Lean *proof step* task is to predict the next tactic given this goal state. We refer the reader to subsection A.1.1 for examples and further explanation.

Our human tactic proof step dataset consists of source-target pairs of strings, one for each tactic invocation in the Lean core library and in `mathlib`. The source string is the pretty-printed tactic state. The target string is the tactic invocation as entered by a human author of the source code. This data is gathered by hooking into the Lean parser and Lean's compilation process. We refer to the task of predicting the next human tactic proof step given a tactic state as the *proofstep objective*.

In this section, we describe the PACT task suite and how data for these tasks are extracted.

For every proof term τ, we record the type Γ of τ, its name `nm`, and a list `ps` of all premises (*i.e.* named references to other lemmas in the library) which are used in τ. We then recurse through τ, tracking a list `bs` of bound variables which we update whenever navigating into the body of a λ-expression. At every subterm $\tau' \subseteq \tau$ we record τ', its type Γ', the current state of `bs`, and the following data:

1. A *tactic state*, where the goal is set to be Γ' and the list of hypotheses in the local context is set to be the list `bs`, *i.e.* those bound variables in scope at τ'.
2. A *partial proof term*, *i.e.* τ with τ' masked out.
3. A *premise selection bitmask*, *i.e.* Boolean labels for every `p` in `ps` indicating whether `p` is used in τ'.
4. A *local context bitmask*, *i.e.* similar Boolean labels for every `b` in `bs` indicating whether `b` is used in τ'.

5. An optional *next lemma*: if the first step of τ' is to apply a premise `p` in `ps`, we record `p`.

Whenever we record a term, we record both *pretty-printed* and far more explicit *fully elaborated* versions of it. The fully elaborated terms explicitly display enormous amounts of type information which are usually silently inferred by Lean. From these data, we assemble the following language modeling tasks:

1. **Next lemma prediction.** Given the tactic state, predict the next lemma to be applied.
2. **Proof term prediction.** Given the tactic state, predict the entire proof term τ'.
3. **Skip-proof.** Given the partial proof term, predict the masked subterm τ'.
4. **Type prediction.** Given the partial proof term, predict the type Γ' of the masked subterm τ'.
5. **Tactic state elaboration.** Given the tactic state, predict the fully elaborated tactic state.
6. **Proof term elaboration.** Given τ, predict the fully elaborated version of τ.
7. **Premise classification.** Given the tactic state and a premise $\texttt{p} \in \texttt{ps}$, predict either `<TRUE>` or `<FALSE>` according to the premise selection bitmask.
8. **Local context classification.** Given the tactic state (which consists of a list of local assumptions `bs` and the goal Γ'), predict the sublist of `bs` which is true on the local context bitmask.
9. **Theorem naming.** Given the type Γ of the top-level proof term τ, predict the name `nm`.

We remark that our next lemma prediction task is precisely the low-level `PROOFSTEP` objective studied in [48], and our skip-proof task superficially resembles, but is much more difficult than the skip-tree task studied in [66], as proof terms tend to be far more complex than the syntax trees of theorem statements.

3.1.2.2 Machine learning environment

We instrument Lean for automatic theorem proving with a language model, including utilities for:

1. setting the runtime environment at a particular theorem (ensuring proofs are never circular),
2. serializing the tactic state as environment observations for a theorem-proving agent,
3. exposing Lean's parser to re-parse strings emitted by a language model into tactic invocations, and
4. executing and capturing the results of the re-parsed tactics, enabling the recording of trajectories for expert iteration and reinforcement learning.

In addition to this general instrumentation, we implement a generic best-first search algorithm for theorem proving; it forms the basis for our evaluations and is written entirely in Lean itself. The algorithm is parametrized by an oracle `(Ω : tactic_state → list (string × float))` that accepts a tactic state and returns a list of strings and heuristic scores. The search is controlled by a priority queue of *search nodes*, which consist of a tactic state (*i.e.* a partial proof) and search metadata. In the outer loop of the algorithm—which continues until either the theorem is completely proved (*i.e.* no goals are remaining on the current node), the priority queue is empty (*i.e.* the search has failed), or a pre-set timeout or budget of iterations is exceeded—we pop a node off the queue, serialize the associated tactic state and use it to query the oracle, producing a list of candidates `cs : list (string × float)`. We then loop over the candidates `cs` to produce a list of new search nodes, by re-parsing each string into a tactic and adding a new node if the parsed tactic advances the proof without raising errors. These new search nodes are then re-inserted into the queue in order of decreasing priority and the search continues. We optionally constrain the search by enforcing maximum width and depth limits $w_{\max}$ and $d_{\max}$ that guard insertion into the queue. When considering nodes for insertion, any node whose depth exceeds $d_{\max}$ is ignored, and all nodes are ignored if the queue size is strictly larger than $w_{\max}$. Due to the flexibility in assigning heuristic scores and in choosing the maximum width and depth hyperparameters, our algorithm is quite general—for example, it reduces to (1) a greedy depth-first search when $w_{\max} = 0$, and (2) a naïve breadth-first search when heuristic scores are identical and $w_{\max} = d_{\max} = \infty$.

3.1.3 Experiments

In all of our experiments, we use decoder-only transformers similar to GPT-3 [23]. Unless mentioned otherwise, all of our models have 24 layers with $d_{\text{model}} = 1536$ and 24 heads, accruing to 837M trainable parameters. They are also pre-trained on `WebMath` [48] for 72B tokens. We use the standard `BPE` encoding [23], a batch size of 512 and a learning rate of 0.00025 with a cosine schedule and a 100-step ramp-up.

We use an 80-5-15 train-validation-test split. We split all datapoints deterministically by *theorem name*, by hashing each name to a float in $(0, 1)$. This ensures, for example, that proof steps used to prove a test theorem never appear in the training data and vice-versa.

When fine-tuning a model we load its saved parameters but re-initialize the optimizer. We start each training for a fixed number of tokens (defining the cosine schedule) and record the number of tokens consumed as we reach a minimal validation loss. We use the minimum validation loss snapshot to evaluate each model on our held-out test set.

We partition our datasets into three groups:

1. `tactic`: the dataset described in subsubsection 3.1.2.1.
2. `mix1`: the union of the PACT tasks **next lemma prediction** and **proof term prediction** (subsubsection 3.1.2.1), selected because of their close relation to `tactic`.
3. `mix2`: all other datasets described in subsubsection 3.1.2.1.

This grouping is motivated by the impossibility to ablate each dataset separately given our compute budget. They nonetheless enable us to study the effect of tasks that are very close to the `tactic` objective in comparison to others. Our choice of **next lemma prediction** and **proof term prediction** for `mix1` is motivated by the observation that these tasks are closely related to the theorem proving objective: a proof can be given entirely in terms of a sequence of lemmas to apply (as in Metamath), or the proof can be finished in one step by supplying the entire proof term. Despite their logical similarity to the `PROOFSTEP` objective, we nevertheless use different keywords in the prompt to the model to disambiguate (`NEXTLEMMA` and `PROOFTERM`) from (`PROOFSTEP`) because the data is noisy and represents a significant distribution shift: during pretty-printing, subtrees of proof terms beyond a certain depth are dropped entirely, there is generally no guarantee that they can be re-parsed, and

the data is much more verbose than what humans typically supply in source code.

We run theorem-proving evaluations on our held-out `test` set, comprising 3071 theorems. Since the split was conducted by theorem name, the proofs of these theorems never appear in the training data. For each theorem in the test set, we set the runtime environment to the location where the theorem is proved in the source code, preventing the use of theorems defined later in `mathlib` and ensuring that we never derive circular proofs. We compare against existing proof automation In Lean by also evaluating the tactics `refl`, which attempts to prove statements via definitional equality, and `tidy`, which conducts a greedy depth-first search using a fixed list of tactics at each step. We re-implement `tidy` as a special case of our best-first search algorithm using an oracle which always emits the same list of tactics, and so henceforth refer to it as `tidy-bfs`. In all of our experiments, we use a maximum width of $w_{\max} = 16$, a maximum depth of $d_{\max} = 128$, a maximum budget of 512 iterations of the outer loop, a timeout of 5 seconds per tactic execution, and a global timeout of 600 seconds per theorem. Because sampling completions from our models is much slower (≈ 1 second) than querying the constant `tidy-bfs` oracle (instantaneous), the `tidy-bfs` search runs many more iterations than `gptf` before timeout.

We report the pass-rate (*i.e.* percentage of theorems proved) from the randomly-chosen held-out test set, following [46], [74], and others. We provide an alternative pass-rate at the end of this section, using theorems added to `mathlib` after our dataset was collected. We average over three evaluation runs when reporting the pass rate.

We first study the effects of pre-training versus co-training with the `mix1` and `mix2` datasets. We pre-train using the methodology described above (potentially pre-training first on `WebMath`, and then on a PACT dataset in sequence). For co-training, we simply concatenate and shuffle the datasets together without applying any particular weight to a given dataset.

The main results are presented in Figure 2. Pre-training exhibits an effective transfer from `mix-1` and/or `mix-2` but the best result is achieved by co-training with both these datasets. With this setup, we are able to train for much longer (71B tokens vs 22B+18B for the best pre-training setup) before overfitting on the `PROOFSTEP` task. We hypothesize that PACT regularizes overfitting to the `PROOFSTEP` task while still imparting useful knowledge to

`tactic`	
tactic proof steps	`GOAL <TacticState> PROOFSTEP <Tactic>`
`mix1`	
next lemma prediction	`GOAL <TacticState> NEXTLEMMA apply (<NextLemma>)`
proof term prediction	`GOAL <TacticState> PROOFTERM exact (<ProofTerm>)`
`mix2`	
skip proof	`RESULT <MaskedProofTerm> SKIPPROOF <ProofTerm>`
type prediction	`RESULT <MaskedProofTerm> PREDICTTYPE <Type>`
tactic state elaboration	`GOAL <TacticState> ELABGOAL <ElaboratedTacticState>`
proof term elaboration	`PROOFTERM <ProofTerm> ELABPROOFTERM <ElaboratedProofTerm>`
premise classification	`GOAL <TacticState> CLASSIFYPREMISE <Premise> <True|False>`
local context classification	`GOAL <TacticState> CLASSIFYLOCALS <LocalsList>`
theorem naming	`TYPE <Type> NAME <Name>`

Figure 1: Auto-regressive objectives used for each task described in subsubsection 3.1.2.1. Placeholders represented with brackets (such as `<TacticState>`) are substituted by the context-completion pairs from each datasets in the prompts above. Each task is presented to the model with its respective keyword (`PROOFSTEP`, `NEXTLEMMA`,...). We wrap the completions of `mix1` tasks (with `apply(...)` and `exact(...)` respectively) as a hint that they are related to the respective Lean tactics; this is not directly possible for the other tasks.

the model due to large amounts of mutual information, and that this is the main driver of increased performance.

Next, we ablate the effect of `WebMath` pre-training (instead starting with a model pre-trained on the same English language mix as GPT-3). As expected, co-trained models suffer from a performance drop without `Webmath` pre-training. but we were more interested in measuring the effect on pre-trained models on `mix-1` and `mix-2`, as they may not benefit from `WebMath` as much due to the two successive pre-training steps.

We report the optimal validation losses in Figure 3. `WebMath` appears as substantially beneficial even in the sequential pre-training setup. This indicates that PACT is not a replacement for `WebMath` pre-training, but rather a complementary method for enhancing the performance of language models for theorem proving.

We rule out the possibility that the benefits from PACT come from simply regularizing our models on the scarce `tactic` data alone. We checked that a `WebMath > tactic` model trained with 15% residual dropout achieved a minimum validation loss of 1.01 and 33.6% pass rate, far below the 48.4% PACT pass rate.

Finally, we study how performance scales with respect to model size. We use the best training setup reported in Figure 2, `WebMath > mix1 + mix2 + tactic`. The `837m` model is our main model. The `163m` and `121m` models respectively have 12 and 6 layers, with $d_{\text{model}} = 768$. The learning rates are respectively adjusted to 0.0014 and 0.0016.

As demonstrated by Figure 4, performance is highly correlated with model size, with larger models generally achieving better generalization even in the overfitted regime. We leave as future work a careful study of how evaluation performance is affected when scaling to multi-billion parameter models, as well as the feasibility of deploying them for interactive use by Lean users.

In the five-week period that separated our last dataset extraction and the writing of the material in this section, `mathlib` grew by 30K lines of code, adding 2807 new theorems. Evaluating our models on these new theorem statements gives a unique way to assess their capability to assist humans in formalizing proofs and to test their generalization to completely unseen theorems and definitions. This evaluation set also addresses one of the weaknesses of using a random split of theorems from a formal mathematics library, namely that the split is

non-chronological; *e.g.* test theorems can appear as lemmas in proofs of train theorems.

We call this temporally held-out test set `future-mathlib` and evaluate our best model as well as the `refl` and `tidy-bfs` baselines on it. In contrast to evaluation on our `test` split, the `refl` baseline (simply attempting a proof by the `refl` tactic) closes 328 proofs (11.6%), demonstrating an important skew towards trivial boilerplate lemmas generally defined to provide alternate interfaces to new definitions. The `tidy-bfs` baseline closed 611 proofs (21.8%), and our best model `wm-tt-m1-m2` closed 1043 proofs (37.1%), proving 94% of the `refl` lemmas. We attribute the weaker performance to heavy distribution shift: by the nature of the dataset, the `future-mathlib` theorems frequently involve new definitions and concepts which the model was never exposed to during training. Nevertheless, the success rate remains high enough to suggest strong generalization and usefulness at the frontier of formalized mathematics.

3.1.4 Discussion

In Lean, multiple tactic commands can be chained together using semicolons. Our data pipeline treats these tactic chains as a single sequence in our training data, and they are occasionally predicted by the model. Such chained tactic applications are difficult for human formalizers to synthesize on their own, as they require reasoning about the semantics of multiple tactics in sequence and their effects on the tactic state, and the examples present in the training data are usually optimized by hand from longer, less succinct proofs. We observed that PACT significantly boosts the capability of our models to *successfully* predict longer chained tactic applications. This occurs despite the fact that the tactic chaining idiom is specific to the tactic proofstep dataset and does not appear in the PACT data whatsoever. We supply more detail in subsection A.3.1.

We also evaluate our best PACT model (`wm-to-tt-m1-m2`) on the theorem naming task, using the theorem statements and human-supplied names from the `future-mathlib` evaluation set. It achieved 20% acc@1, 27% acc@10, and 30% acc@16. (Here, acc@N means the estimated likelihood that at least one out of N attemped independent tries is an exact match to the ground truth.) An inspection of its outputs reveals that even when its

predictions diverge from the ground truth, they are often idiomatic and semantically correct alternatives. We supply more detail in subsection A.3.2.

Lean's `mathlib` [61] is a rapidly growing open source library of formal mathematics which has grown considerably in size each year for the past four years.[1] Our work has been welcomed by members of this community, with Lean power users describing some of the new proofs found by GPT-f as "nontrivial" and "clever". More than one-third of the proofs found by our models are shorter and produce smaller proof terms (sometimes by several orders of magnitude) than the ground truth. Manually inspecting a small, non-cherry picked sample of these shorter proofs has led to nineteen GPT-f co-authored commits to `mathlib`, some of which reduce proof term sizes and theorem compilation times by an order of magnitude (see section A.4).

There are many elaborations on the training data, training methodology, and tree search wrapping `lean-gptf` which can be reasonably expected to improve its performance at theorem proving. Our dataset can be synthetically augmented using similar methods as [48]. Our dataset could be cleaned further, and proofs minimized. Merely making the decoded rewrites robust by only using the largest prefix of successful rewrites significantly boosts the success rate of suggested rewrites. In a similar vein, predicted lemmas generated as arguments to unsuccessful tactic applications could be cached and re-used as hints for an intermittently-queried hammer. The increased success rate of chained tactic predictions mentioned above shows the feasibility of having language models perform multiple reasoning steps in a single query, potentially improving the efficiency of the proof search. From the experiments described in subsection 3.1.3, it is clear that the composition of the dataset used for co-training significantly affects performance on theorem proving. Although we uniformly sampled across all co-training tasks, it would be interesting to optimize a dynamic mixture schedule, perhaps annealing towards a desired task.

[1] See `https://leanprover-community.github.io/mathlib_stats.html` for up-to-date statistics on `mathlib`'s size and growth over time.

3.1.5 Acknowledgments

We thank the members of the Lean community, in particular Kevin Buzzard, Simon Hudon, Johan Commelin, Mario Carneiro, Bhavik Mehta, and Gabriel Ebner for their valuable feedback on our work. We are indebted to Markus Rabe and Christian Szegedy for many hours of helpful discussion. We also thank Daniel Selsam, Tom Hales, and Josef Urban for feedback on earlier drafts of the material in this section.

3.2 Synthetic proof term data augmentation

Imitation learning for the task of formal theorem proving is bottlenecked by the limited size of existing libraries of formalized mathematics. We propose SPT-Aug (Synthetic Proof Term Augmentation), a methodology for using samples from trained language models in conjunction with the Lean kernel to generate novel training examples. In particular, we train language models to generate Lean proof terms unconditionally, then we sample from such models to generate collections of proof term candidates. Using the Lean kernel to identify typecorrect (*i.e.* successfully typecheckable using the Lean kernel) proof term candidates and infer corresponding types, we create novel training examples for conditional and unconditional proof term language modeling. When used to augment existing datasets, these generated training examples are shown to improve the perplexity of both conditional and unconditional proof term language modeling on a held-out test set.

Some material in this section appears in [75], published in the proceedings of AITP 2022, and the work in this section is joint with Joseph Palermo and Johnny Ye.

3.2.1 Introduction and background

We remind the reader that a Lean "proof term" is a representation of a mathematical proof. Lean proof terms are represented with a datatype called an expression. Expressions are inductively defined and can be traversed with tree-traversal algorithms. The Lean kernel provides a mechanism to verify that provided expressions are well-formed and type-correct.

In the case that provided expressions are well-formed and type-correct, Lean also provides a mechanism to infer their type. Note that by the Curry-Howard correspondence, the type of a proof is effectively a theorem statement for that proof [76]. Lean expressions can be converted into pretty-printed strings using utilities provided by the Lean metaprogramming framework [77]. Other utilities enable parsing pretty-printed strings back into expressions. In some cases cases, a pretty-printed string may lack sufficient context to be parsed back into a Lean expression due to insufficiency of the pretty-printing mechanism (todo: cite). However, in practice pretty-printed strings are still a useful representation of Lean expressions and have been used successfully as a representation for training language models [49]. The Lean mathematical standard library mathlib is the primary repository of formalized mathematics in Lean [78]. It contains a growing repository of tens of thousands of theorems and lemmas across a diverse range of mathematical domains including algebra, analysis, geometry, probability theory, and many others. It is used in this work as the basis from which to construct datasets for training language models.

3.2.1.1 Comparison to related work

In section 3.1, we demonstrate how to construct many auxiliary tasks (not directly related to the theorem proving task) for language model training and showed that jointly training on these additional tasks lead to transfer learning for the task of theorem proving. INT [29] procedurally generates synthetic inequality-based theorems to systematically investigate out-of-distribution generalization. By contrast, in our work we use a learned model along with the Lean kernel to generate arbitrary proofs and theorems in Lean's higher order logic. A recent survey [79] provides an overview of the subfield of data augmentation strategies for natural language processing and classifies its methods into the categories: rule-based, example interpolation, and model-based. Our work is most similar to other model-based techniques such G-DAUG [80], which involves generating synthetic examples using pre-trained language models, and selecting according to several criteria the most useful candidate examples to use for fine-tuning. Our work is distinguished by the use of unconditional sampling to generate examples for both conditional and unconditional modelling and also the use of the Lean

kernel to filter synthetic examples.

3.2.2 Methodology

3.2.2.1 Bootstrap Datasets

In order to use trained language models to generate new candidate training examples we must first create an initial "bootstrap" dataset from which a first model can be trained. Our examples for conditional proof term language modelling have the form:

THEOREM <theorem> PROOF <proof> EOT

By contrast our examples for unconditional proof term language modelling have the form:

PROOF <proof> EOT

Thus to generate conditional examples we require theorem-proof pairs while to generate unconditional examples we require only proofs. We represent theorems and proofs as pretty-printed strings.

We extract a dataset of proofs and corresponding theorems by processing the declarations in mathlib. Each declaration in mathlib has a value and a type. By the Curry-Howard correspondance [76], a value corresponds to a proof and a type corresponds to a theorem statement for the proof.

Proof expressions can be traversed to identify unique sub-expressions which, themselves, are also valid proofs[81]. We use such a traversal to extract a larger dataset than would otherwise be obtained. This is implemented as a depth-first tree traversal: a proof expression is provided as input. However, before running the traversal on an expression, we ensure that, when pretty-printed, it has a length less than 4096 characters. This is an optional step that reduces the run-time of the data processing pipeline and, also, prevents the sub-proofs of extremely long proofs from dominating the dataset.

At each step in the traversal we 1) test if the expression is a valid proof by using tactic.is_proof, 2) we convert it into a pretty-printed string, 3) we ensure the length of the pretty-printed string is below the required threshold of 4096 characters. If conditions 1 and 3 are met, the pretty-printed string and its type universe variables are serialized for downstream processing.

After serializing the proof strings and the corresponding type universe variables, we replace the type universe variables in the proof strings with a canonical set which do not appear in mathlib (i.e. u_1001, u_1002, etc...). The purpose of this is to simplify later parsing, as our parser requires specifying the set of type universe variables up-front.

Subsequently, we parse and typecheck all the serialized proofs to ensure that pretty-printing has not rendered them invalid. If the typecheck passes then we also extract the proof's type (i.e. theorem statement) from the Lean kernel's output and serialize that as well. Finally, we construct conditional and unconditional training examples from the serialized theorem-proof pairs.

3.2.2.2 Creating Synthetic Datasets

Once a language model has been trained on a dataset of unconditional training examples then it can be prompted with "PROOF" to induce it to generate a synthetic proof. We sample from such language models repeatedly to generate proof candidates. After accumulating a large batch of proof candidates we parse and typecheck them, and if the typecheck is successful we also extract the corresponding type. The output of this process are theorem-proof pairs which can be used to produce additional synthetic examples of the form described in section 3.1.

In addition to examples that match the format of the bootstrap dataset examples, we also generate a new type of example from proof candidates regardless of whether or not they have passed typechecking:

NON_TYPECHECKED_PROOF <non_typechecked_proof> EOT

These examples are useful in evaluating the relative utility of examples that have been verified to be type-correct examples vs. examples that are merely type-correct with some probability. In other words, these examples are useful in determining the importance of the typecheck filter provided by the Lean kernel for improving the quality of the synthetic data.

3.2.3 Experimental Setup

3.2.3.1 Splitting the Bootstrap Data

We split our bootstrap dataset into train, validation and test sets. We do so firstly by splitting on declaration names. However, we found it was also necessary to apply a further filter to our validation and test sets to remove examples deemed too similar to training examples. The reason for this is that although declarations are themselves unique, the sub-proofs extracted from them which comprise the source for the training examples are not necessarily unique and may be shared across declarations either in whole or in part.

To apply a further filter, we use cosine similarity with a TF-IDF [82] embedding to find the most similar training example to each validation/test example. Note that the the most similar training example is whichever training example maximizes cosine similarity to the validation/test example in question. We then compute the Levenstein distance between each validation/test example and the most similar example in the training set. Thus, we remove any validation or test example (x_{test}) for which:

$$\frac{levenstein\_distance(x_{test}, \arg\max_{x_{train}} cosine\_similarity(x_{test}, x_{train}))}{length(x_{test})} < threshold \quad (1)$$

In our experiments we set the threshold to 0.15. This setting was determined after we found empirically that relaxing the threshold led to overfitting.

Our initial split of declaration names into train:validation:test sets is done at a ratio of 650:175:175, producing 207,194 train examples, 55,470 validation examples and 54,964 test examples. After the additional filtering step, 11,145 validation examples and 10,233 test examples remain. Note that each example can be represented in either conditional or unconditional form since a theorem-proof pair is available for each. For reference, note that we name the bootstrap training sets as follows:

- **cond_train_bootstrap**: bootstrap dataset containing conditional training examples
- **uncond_train_bootstrap**: bootstrap dataset containing unconditional training examples

3.2.3.2 Proof Term Language Modelling Experiments

For these experiments, we train language models using fairseq [83]. We utilize fairseq's implementation of GPT2 [22] with approximately 2B parameters (the so-called "big" size). We also utilize fairseq's implementation of the "gpt2" byte pair encoder. We set max-tokens to 1536 and batch size to 512. We optimize parameters using SGD with a fixed learning rate of 0.01 and set patience to 100 epochs. We use a default value of 0.1 for dropout.

We train baseline models using the cond_train_bootstrap and uncond_train_bootstrap datasets. Recall that we use models trained on unconditional datasets to generate proof candidates. In particular we use the baseline model trained on uncond_train_bootstrap to sample over 20M unique proof candidates.

We sample proof candidates using beam search [84]. The temperature hyperparameter used for beam search is inversely proportional to typecheck pass rate and similarity of type-correct generated proofs to the training set (see Figure 1). Ideally we would want a high typecheck pass rate and low similarity of generated examples to the training set. Thus there is a trade-off to navigate between typecheck pass rate and diversity in the generated data. It's unclear how to navigate this tradeoff without generating many datasets with different parameter settings and comparing their relative utilities. For this study we set temperature to 1.3 and use a beam size of 5.

From the set of unique generated candidates, 1.57 % or 352,469 unique proofs passed typecheck. Note that typecheck pass rate is also effected by the number of unique proof candidates generated because the greater the number of previously generated proof candidates the more unlikely a proof candidate needs to be in order to be unique.

By combining the original bootstrap datasets (namely, `cond_train_bootstrap` and `uncond_train_bootstrap`) with these generated examples we construct new augmented datasets. These augmented datasets are controlled for size by augmenting either by # of additional examples or # of additional characters.

- **cond_train_aug**: cond_train_bootstrap with +100% conditional typechecked (weighted by # of additional examples)
- **uncond_train_non_tc_aug**: uncond_train_bootstrap with +100% unconditional

non-typechecked (weighted by # of additional characters)

- **uncond_train_half_tc_aug**: uncond_train_bootstrap with +50% unconditional non-typechecked and +50% unconditional typechecked (weighted by # of additional characters)
- **uncond_train_tc_aug**: uncond_train_bootstrap with +100% unconditional typechecked (weighted by # of additional characters)

Note that we weight the additional unconditioned examples by counting characters because the unconditioned examples comprise both typechecked and non-typechecked proofs, and the average length of non-typechecked proofs tends to be longer. In particular we find an average length of 275 characters for non-typechecked proof candidates vs. an average length of 162 characters for proofs that passed typecheck (see appendix for more details).

In both the conditional and unconditional case we find that training on the augmented datasets results in lower final loss. In the unconditional case we find that better loss is achieved for training sets in which a higher percentage of the augmented data is type-correct. This demonstrates the utility of the Lean kernel in filtering out invalid samples before training. However, it's important to keep in view the fact that since only a small percentage of synthetic proofs pass typecheck, in practice we can expect the best results to be achieved by simply training on all generated examples since such a dataset would be much larger.

We investigate how much of the improvement in loss associated with training on an augmented dataset is accounted for by an increase in regularization that can be achieved with dropout. To do this we train models on cond_train_bootstrap and cond_train_aug with successively higher levels of dropout (incrementing by 0.1), until increasing dropout no longer improves the best achieved validation loss.

We find that while test performance is improved by increasing dropout from our baseline value of 0.1, optimal performance is achieved at a value of 0.3. Notably, even with optimized dropout we observe a significant performance advantage from training on the augmented dataset.

Model	Tokens elapsed	`mix1`	`mix2`	`tactic`	Pass-rate
Baselines					
`refl`					1.1%
`tidy-bfs`					9.9%
`WebMath > tactic`	1B			1.02	32.2%
Pre-training					
`WebMath > mix1`	11B	*0.08*			
`WebMath > mix2`	16B		*0.08*		
`WebMath > mix1 + mix2`	22B	*0.11*	*0.08*		
`WebMath > mix1 > tactic`	1B			1.00	39.8%
`WebMath > mix1 + mix2 > tactic`	1B			0.97	44.0%
Co-training (PACT)					
`WebMath > mix1 + tactic`	18B	*0.08*		0.94	40.0%
`WebMath > mix2 + tactic`	75B		*0.09*	0.93	46.0%
`WebMath > mix1 + mix2 + tactic`	71B	*0.09*	*0.09*	**0.91**	**48.4**%
Pre-training and co-training					
`WebMath > mix2 > mix1 + tactic`	18B	*0.08*		0.93	46.9%

Figure 2: Comparison of pre-training and co-training on `mix-1` and `mix-2`. We use `>` to separate distinct pre-training phases and `+` associatively indicates that a co-training mixture. As an example, `WebMath > mix2 > mix1 + tactic` signifies a model successively pre-trained on `WebMath` then `mix2` and finally co-trained as a fine-tuning step on `mix1` and `tactic`. Columns `mix1`, `mix2`, `tactic` report the optimal validation loss achieved on these respective datasets. We provide a detailed description of experiment runtime and computing infrastructure in section A.2. "Tokens elapsed" means the number of tokens the model was trained on.

Model	Tokens budget	Tokens elapsed	`mix1`	`mix2`	`tactic`	Pass-rate[†]
Baselines						
`tactic`	32B	1B			1.59	—
Pre-training						
`mix1`	32B	20B	*0.12*			
`mix2`	32B	25B		*0.10*		
`mix1 + mix2`	32B	27B	*0.13*	*0.10*		
`mix1 > tactic`	32B	1B			1.26	—
`mix1 + mix2 > tactic`	32B	1B			1.16	—
Co-training						
`mix1 + tactic`	32B	27B	*0.11*		1.12	—
`mix2 + tactic`	96B	75B		*0.10*	**1.02**	40.4%
`mix1 + mix2 + tactic`	96B	71B	*0.10*	*0.11*	1.07	—
Pre-training and co-training						
`mix2 > mix1 + tactic`	32B	26B	*0.11*		1.09	—

Figure 3: Validation losses achieved in the pre-training and co-training setups without `WebMath` pre-training. See Figure 2 for a description of the columns and the models nomenclature used. [†]Due to technical constraints, we are unable to provide pass-rates for some of the models. We use `>` to separate distinct pre-training phases and `+` associatively indicates that a co-training mixture. "Tokens elapsed" means the number of tokens the model was trained on.

Model	Tokens budget	Tokens elapsed	`mix1`	`mix2`	`tactic`	Pass-rate
`121m`	96B	82B	*0.13*	*0.10*	1.23	35.1%
`163m`	96B	80B	*0.12*	*0.09*	1.11	39.8%
`837m`	96B	71B	*0.09*	*0.09*	**0.91**	**48.4**%

Figure 4: Validation losses and pass-rates achieved for various model sizes using PACT. See Figure 2 for a description of the columns. The setup used is `WebMath > mix1 + mix2 + tactic`. We use `>` to separate distinct pre-training phases and `+` associatively indicates that a co-training mixture. "Tokens elapsed" means the number of tokens the model was trained on.

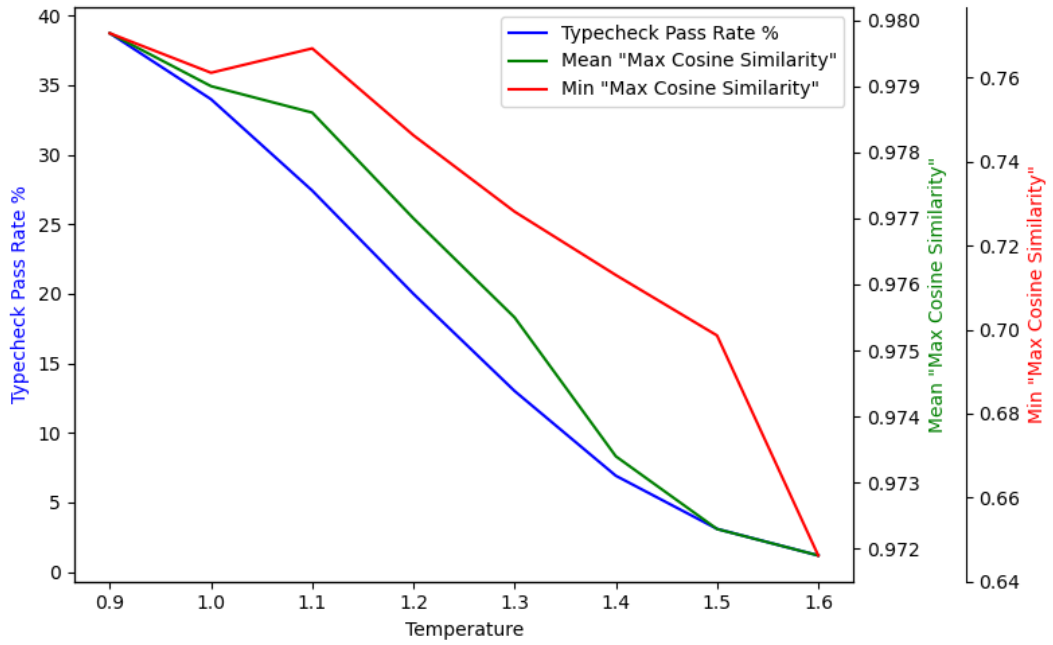

Figure 5: As sampling temperature is increased, both typecheck pass rate and similarity of each type-correct generated example to the most similar training example (i.e. "max cosine similarity") decreases. Note that the mean "max cosine similarity" is the average observed max cosine similarity across the type-correct generated examples with a given temperature.

Dataset	Test Loss	Test Perplexity	Test Accuracy
cond_train_bootstrap	1.252	2.38	9.72%
cond_train_aug	**1.122**	**2.18**	**16.92**%
uncond_train_bootstrap	1.744	3.35	N/A
uncond_train_non_tc_aug	1.721	3.30	N/A
uncond_train_half_tc_aug	1.714	3.28	N/A
uncond_train_tc_aug	**1.701**	**3.25**	N/A

Table 1: Test loss, test perplexity, and test accuracy (if appropriate) of the best model checkpoint trained on each dataset.

	Dropout: 0.1	Dropout: 0.2	Dropout: 0.3	Dropout: 0.4
Bootstrap Loss	1.252	1.147	1.091	1.088
Augmented Loss	1.122	1.043	**1.008**	1.013
Loss Gap	0.13	0.104	0.083	0.075
Bootstrap Perplexity	2.38	2.21	2.13	2.13
Augmented Perplexity	2.18	2.06	**2.01**	2.02
Perplexity Gap	0.2	0.15	0.12	0.11
Bootstrap Accuracy	9.72%	11.47%	12.97%	14.2%
Augmented Accuracy	16.92%	18.29%	**20.82**%	19.37%
Accuracy Gap	7.2%	6.82%	7.85%	5.17%

Table 2: Test loss, test perplexity, and test accuracy of the best model checkpoint trained on each dataset with different levels of dropout. "Bootstrap X" rows indicate test performance after training on cond_train_bootstrap. "Augmented X" rows indicate test performance after training on cond_train_aug. "X Gap" rows are computed by taking the absolute difference between "Augmented X" and "Bootstrap X".

4.0 Statement curriculum learning and applications to Olympiad theorem proving

In this chapter, we explore the use of expert iteration in the context of language modeling applied to formal mathematics. We show that, at the same compute budget, expert iteration– by which we mean proof search interleaved with learning–dramatically outperforms proof search only. We also observe that, when applied to a collection of formal statements of sufficiently varied difficulty, expert iteration is capable of finding and solving a curriculum of increasingly difficult problems, without the need for associated ground-truth proofs. Finally, by applying this expert iteration to a manually curated set of problem statements, we achieve state-of-the-art on the *miniF2F* benchmark, automatically solving multiple challenging problems drawn from high school olympiads.

Some material in this section appears in the paper [85], which was submitted to ICLR 2022. The work in this section is joint with Stanislas Polu, Kunhao Zheng, Mantas Baksys, Igor Babuschkin, and Ilya Sutskever.

4.0.1 Introduction

Deep learning has found success in many domains, such as language [86, 87, 88], vision [89, 90], and image generation [91, 92]. However, there is one domain where deep learning has not yet enjoyed a comparable success: tasks that require *planning* and *symbolic reasoning*. The exception to this are two-player games [93, 94, 95, 96], in which deep learning systems exhibit a considerable degree of reasoning, especially when trained with self-play combined with a search procedure, such as Monte Carlo Tree Search (MCTS) [97]. Even so, the reasoning abilities achieved are limited due to the relatively narrow scope of games.

Theorem proving in interactive proof assistants, or formal mathematics, appears as an interesting game-like domain to tackle due to its increased scope. Like games, formal mathematics has an automated way of determining whether a trajectory (*i.e.* a proof) is successful (*i.e.* formally correct). However, the vast scope of formal mathematics means that

any strong reasoning result obtained in it will be more meaningful than comparable results in games (*e.g.* finding proofs to mathematical conjectures), and could even be applicable to important practical problems (*e.g.* software verification).

However, tackling formal mathematics involves two main challenges that we must address in order to continue making progress:

- **Infinite action space:** not only does formal mathematics have an extremely large search space (like Go for example), it also has an infinite action space. At each step of proof search, the model must choose not from a well-behaved finite set of actions, but a complex and infinite set of tactics, potentially involving exogenous mathematical terms that have to be generated. For instance, generating a mathematical statement to be used as a witness[1] or a cut[2].
- **No direct self-play setup** In formal mathematics, a prover is not playing against an opponent but against a set of statements to prove. When faced with a statement that is just too hard, there is no obvious reframing of the formal mathematics setup that will let the prover generate intermediary easier statements to tackle first. This asymmetry prevents naive application of the symmetric self-play algorithms commonly used in 2-player games.

There are two main differences between reinforcement learning and formal mathematics that make a naive application of the former to the latter unlikely to succeed. Past work proposed to address the infinite action space problem by sampling from a language model [3]. This section focuses on the second problem and our basis for addressing it is the observation that the key role of self-play is to provide an unsupervised curriculum. We propose instead to supply auxiliary sets of problem statements (without requiring proofs) of varying difficulty. We empirically show that, when the difficulty of these auxiliary problems is varied enough, a simple expert iteration procedure is able to solve a curriculum of increasingly difficult problems, eventually generalizing to our target distribution. We show that this works with both automatically-generated and manually-curated auxiliary distributions of problems and leverage this to achieve state-of-the-art on the *miniF2F* benchmark. Our results suggest

[1] A witness is an object used in certain "there exists an x ..." proofs.

[2] A cut is the introduction and the chaining of a lemma in the middle of a proof.

that continuous self-improvement in formal mathematics can potentially be reduced to the problem of generating such sets of formal statements, which we have done in part manually in this work, but could eventually be scaled in the future with more automation (such as more domain-specific statements generator or even informal to formal machine translation).

4.0.1.1 *miniF2F* benchmark

The reason we target the *miniF2F* [2] benchmark in this work is threefold. The benchmark consists of 244 *validation* and 244 *test* formalized statements of mathematical problems from various competitions.

- We believe this benchmark to be a better measure of mathematical reasoning compared to a library-derived split, which, due to how formalization is done by humans, generally includes numerous technical lemmas which are not necessarily reasoning-intensive.
- Restricting the problem domain to competition mathematics allow us to work against a fixed and self-contained knowledge base generally well-covered by existing formal libraries. Solving these problems generally do not require the introduction of new mathematical concepts.
- Proof for problems from math competitions are extremely scarce in formal mathematics libraries (which focus on generic theorems and lemmas rather than specific problem statements), which makes *MiniF2F* a challenging benchmark for neural theorem provers and an ideal test-bed for the expert iteration methodology studied in this section.

4.0.1.2 Contribution

Our contributions are the following:

- We present `lean-gym`, a simple REPL interface for interacting with the Lean theorem prover.
- We propose an expert iteration methodology for GPT-f [3] which uses proofs generated by our models as training data to iteratively improve their performance.

- We demonstrate that, at fixed compute budget, expert iteration outperforms proof search only.
- We present a synthetic inequality generator and study how expert iteration finds and solves a curriculum of increasingly difficult problems from a set of generated statements of various difficulty.
- We present a manually curated set of formalized problem statements and leverage it to achieve state-of-the-art performance on the *miniF2F* benchmark.

4.0.2 Related Work

Our work strongly relies on, and can be seen as a natural continuation of the work presented in the original GPT-f paper [3] which studies the use of language models to generate tactics, the PACT paper [49] which applies GPT-f to Lean and studies the benefits from co-training on self-supervised objectives, and the *miniF2F* benchmark [2].

Early applications of deep learning to formal mathematics focused primarily on premise selection and proof guidance. DeepMath [98] explored the use of CNNs and RNNs to predict whether a premise is useful to demonstrate a given conjecture. Their results were later improved with FormulaNet [99] by the use of graph neural networks, reminiscent of NeuroSAT [100]. Proof guidance consists in selecting the next clause to process *inside* an automated theorem prover. Loos et al. [101] investigated the use of models similar to DeepMath's for proof guidance and demonstrated a significant uplift on the Mizar library. More recently [102] demonstrated the potential of deep learning techniques to be competitive with E prover's heuristics when applied to resolution calculus while training on fully synthetic data.

Early applications of deep learning to formal mathematics focused primarily on premise selection and proof guidance. DeepMath [98] explored the use of CNNs and RNNs to predict whether a premise is useful to demonstrate a given conjecture. Their results were later improved with FormulaNet [99] by the use of graph neural networks, reminiscent of NeuroSAT [100]. Proof guidance consists in selecting the next clause to process *inside* an automated theorem prover. Loos et al. [101] investigated the use of models similar to

DeepMath's for proof guidance and demonstrated a significant uplift on the Mizar library. More recently [102] demonstrated the potential of deep learning techniques to compete with E prover's heuristics when applied to resolution calculus while training on fully synthetic data.

HOList [74] proposes a formal environment based on HOL Light. They achieve their best performance [40] with a GNN model designed for premise selection and the use of exploration. The same team studied the use of a skip-tree objective with transformers on formal statements [103], demonstrating, along with GPT-f [3], the potential of leveraging transformers for formal reasoning. *GamePad* [104] and *CoqGymn/ASTactic* [105] introduce environments based on the Coq theorem prover. *ASTactic* generates tactics as programs by sequentially expanding a partial abstract syntax tree. Urban et al. [106] studied the capability of GPT-2 to produce useful conjectures for the Mizar library and IsarStep [107] explored the synthesis of intermediate propositions in

4.0.3 Formal Environment

We choose Lean [55, 108] as our formal environment. Lean benefits from high-level tactics which were shown to be beneficial in the context of the *miniF2F* benchmark [2]–Lean proofs are typically 10x shorter than Metamath's. Also, Lean has recently received a lot of attention from the mathematical community, thanks to projects such as the Perfectoid Spaces [109] and the Liquid Tensor experiment [110], and benefits from a vibrant community of hundreds of contributors to its main mathematical library called *mathlib*. We refer to the PACT paper's Background section [1] for a detailed introduction to Lean in the context of neural theorem proving.

4.0.3.1 `lean-gym`

In the PACT paper [1], proof search is performed by the Lean runtime using the LeanStep environment, with a generic backend interface to models. While easy to use–one just needs to plug in their model–this approach makes it difficult to alter and iterate on the search procedure. This is because it is programmed in Lean (which is not designed or

intended for cluster-wide parallelized I/O intensive tasks), and the coupling of the search procedure with the Lean runtime introduces challenges when scaling to a large number of parallel workers.

To solve these issues we implemented `lean-gym`[3] – a simple REPL interface over the standard input/output implemented in Lean directly. `lean-gym` presents the following API:

- `init-search`: $declaration \rightarrow tactic\_state$. Takes a declaration name (a theorem name from the loaded library) and initializes a search while setting the run-time environment at that particular declaration. It returns the initial tactic state along with a fresh `search_id` and `tactic_state_id`.
- `run_tac`: $(tactic\_state, tactic) \rightarrow tactic\_state$. Takes a `search_id` and a `tactic_state_id` to identify a tactic state, as well as a tactic string to apply to it. It returns a new tactic state and its associated `tactic_state_id`.

Using `lean-gym` is virtually equivalent to opening a Lean editor at a specific theorem, deleting its proof and interacting with Lean to reconstruct it. Appendix subsection A.5.1 provides an example in-terminal execution trace.

Providing a REPL interface over the standard input/output makes it very easy to integrate `lean-gym` from any programming language. Writing a wrapper in Python, as an example, only takes a few dozen lines of code. Since `lean-gym` is a Lean program, managing the loaded libraries is done directly using Lean's own infrastructure (using `leanpkg.toml`), making it quite straightforward to have access to both *mathlib* and *miniF2F* statements from the same `lean-gym` instance.

Note that `lean-gym` is stateful, meaning that distributing proof searches on multiple `lean-gym` instances requires tracking which instance is associated with which proof search. In practice, we were able to scale the use of `lean-gym` to thousands of cores running thousands of proof searches in parallel. Finally, `lean-gym`'s REPL interface is blocking, preventing inner-proof search parallelization, though this limitation can probably be removed in the future.

[3] `https://github.com/openai/lean-gym`

4.0.3.2 Proof datasets extraction

We rely on the proof extraction methodology presented in the PACT paper [1] to extract human tactic proof steps from *mathlib* (the `tactic` dataset) as well as the various other proof artifacts (`mix1` and `mix2` datasets). We also extract *mathlib-{train,valid,test}*, the set of statements from *mathlib* along the split proposed in [1] (the *validation* and *test* splits of `tactic, mix1, mix2` being aligned with *mathlib-{valid, test}* as the splits are determined by declaration name hashes (across all data sources including proof-term mining) as opposed to individual proof steps or data-points).

4.0.4 Expert Iteration

Expert iteration was introduced in [94] and broadly consists in iteratively training models on improvements of their previously sampled trajectories, to achieve continuous improvement. In this section we present our expert iteration methodology, including the models and pre-training strategies we rely on.

4.0.4.1 Model

We use decoder-only transformers similar to GPT-3 [86]. Throughout this section we focus on a model with 36 layers and 774 million trainable parameters (referred to as the *700m* model in the GPT-f paper [3]).

4.0.4.2 Pre-Training

We pre-train our models successively on GPT-3's post-processed version of CommonCrawl (for 300B tokens) and an updated version of *WebMath* [3] (for 72B tokens) whose mix is presented in table Table 3.

As demonstrated in table Table 3, we empirically up-weighted (compared to their token size) parts of *WebMath* with high-quality mathematical content while making sure they don't overfit (despite running more than one epochs for some of them). We also included

Dataset	Size	Mix
Github Python	179 GB	25%
arXiv Math	10 GB	25%
Math StackExchange	2 GB	25%
PACT `mix2`	28 GB	17%
Math Overflow	200 M	5%
ProofWiki	30 M	2%
PlanetMath	25 M	1%

Table 3: Mix and source of data involved in the updated *WebMath* pre-training.

PACT `mix2` directly in the *WebMath* pre-training to avoid having to sequence more than two pre-training phases to prepare Lean models.

4.0.4.3 Training objectives

We rely on two conditional language modeling objectives to train our models. The *proofstep objective*, introduced in [3], consists in generating a `PROOFSTEP` (a Lean tactic) given a `GOAL` (a Lean tactic state). We also condition this objective on the current `DECLARATION` (a Lean theorem name), which remains the same throughout a proof search.

```
DECL <DECLARATION> GOAL <GOAL> PROOFSTEP <PROOFSTEP>
```

The rationale for conditioning on the declaration name is to hint our models on the position of the current declaration in the *mathlib* library. It can be considered as a weak proxy signal for the large amount of information not shown to the model (the full environment consisting of the available imports and currently open declarations such as module names, notations, declared instances, etc.). The declaration name lets models at least in principle memorize and then retrieve some of that information, knowing that `lean-gym` raises an error

if a theorem or definition that is not available in the environment associated with the current declaration is used by tactics generated by our models. Also note that conversely to [3] and like [1] `GOAL` is not necessarily a single goal but a Lean tactic state, which possibly comprises multiple goals.

We depart from [3] and use a *proofsize objective* to guide our proof searches, which consists in generating one token that represents a proof size estimate bucket for the current goal (Lean tactic state).

```
DECL <DECLARATION> GOAL <GOAL> PROOFSIZE <PROOFSIZE_BUCKET_TOKEN>
```

For a given goal g, either the goal was proved as part of the proof search and we denote its proof size (the number of tactic applications (compounded Lean tactics counting as one)) as $ps(g)$, or the goal was not proved in which case we assign the goal to a bucket that virtually represents "infinite" proof sizes.

We use eleven buckets $B = 0...10$ and compute the *proofsize* bucket $b(g)$ for a goal g by assigning infinite proof sizes to bucket 0, all proof sizes over 20 to bucket 1 and linearly projecting proof sizes lower than 20 on the remaining buckets $2, ..., 10$ (10 being the bucket for the shortest proof sizes). In practice, when training and sampling from the model, we map B to the tokens 'A'...'K'.

To value goals as we run proof searches, we sample the *proofsize* bucket token and record the logits $p_b(g)$ for each viable bucket and use them to get a weighted average with the following formula:

$$v(g) = \frac{1}{\#B} \sum_{b \in B} p_b(g).b$$

As an example, if the model assigns $p_0 = 1$ (hence $p_{b \neq 0} = 0$) then $v(g) = 0$. Conversely if the model assigns $p_{10} = 1$ (10 being the bucket for the shortest proof sizes) then $v(g) = 1$.

The rationale for using this *proofsize objective* instead of the *outcome objective* described in [3] is that (1) it achieves better performance compared to the *outcome objective* (see table Table 4), and (2) it prioritizes goals that potentially lead to shorter proofs during proof search, creating an intrinsic incentive for the system to converge towards shorter proofs. Similarly to [3] we favor this token-based approach to the introduction of a separate value head

to keep the overall architecture simple. This way the *proofsize objective* can be implemented by simply augmenting the training dataset and without any architectural change.

4.0.4.4 Bootstrapping

Bootstrapping consists in the steps required to train an initial model on both the *proofstep objective* and the *proofsize objective*.

Given a pre-trained model on *WebMath*, we fine-tune it on the `tactic` dataset extracted from *mathlib* as well as the proof artifacts dataset `mix1` as described in [1]. This initial model, which we denote θ_0 is solely trained on the *proofstep objective*. We use the *validation* splits of the `tactic` and `m1` datasets to early-stop training. Note that this is our only use of *mathlib-valid* to influence the training process throughout this section.

To generate data for the *proofsize objective*, we use θ_0 to sample proofs for statements from *mathlib-train*. For each statement from *mathlib-train* (25k) we attempt $a = 1$ proof searches using the cumulative logprob priority search described in [3] (which does not require a trained value function) using $d = 512$ expansions and $e = 8$ samples per expansion. We denote the set of successful proof searches created in this process as S_0.

Using S_0 we generate dataset D_0 by concatenating:

- The initial `tactic` dataset (*proofstep objective*)
- A deduplicated set of proofsteps extracted from the proofs in S_0 (*proofstep objective*)
- A deduplicated set of proofsize tuples (goals and proofsize) extracted from the full proof searches in S_0 (*proofsize objective*)

Note that the full proof searches in S_0 include goals that are visited but eventually remain unproved, which provides useful negative examples for the trained value function (even if these negatives may include provable goals that simply were not prioritized by the search). Also note that S_0 doesn't include failed proof searches (which would contain only negative examples and no *proofstep objective* data).

We fine-tune θ_0 on D_0 for exactly one epoch (no use of *validation* data for early-stopping) to obtain our initial model θ_1 trained on both the *proofstep objective* and the *proofsize objective*.

θ_0 is used in our expert iteration setup as base model to fine-tune from at each iteration, and θ_1 is our first iterated model or *mathlib* bootstrapped model trained on both objectives.

Model	d	e	*pass@1*	*pass@8*
mathlib-valid				
PACT [1]	512	16	48.4%	
θ_0 ([1] setup)	512	16	48.5%	57.6%
θ_0	512	8	46.7%	57.5%
θ_1	512	8	**56.3%**	**66.3%**
θ_1 (*outcome objective* [3])	512	8	55.6%	65.9%
miniF2F-valid				
PACT [2]	128	16	23.9%	29.3%
θ_0 ([2] setup)	128	16	27.6%	31.8%
θ_0	512	8	28.4%	33.6%
θ_1	512	8	**28.5%**	**35.5%**
θ_1 (*outcome objective* [3])	512	8	28.3%	34.7%

Table 4: Performance of θ_0 and θ_1 on *mathlib-valid* and *miniF2F-valid* compared to PACT Lean GPT-f as reported in [1, 2]. All models have the same architecture. θ_0 is sampled using cumulative logprob priority best-first search. θ_1 is sampled using best-first search based on the *proofsize objective*. We report our setup ($d = 512$ expansions and $e = 8$ tactic samples per expansions) as well as the setups used in [1, 2] to control for compute. We also report the performance of θ_1 on *mathlib-valid* when trained using the *outcome objective* from [3] as an ablation of our proposed *proofsize objective*.

We report in Table 4 the pass rates of θ_0 and θ_1 on *mathlib-valid* and *miniF2F-valid* and compare with previously reported pass rates for equivalent amounts of compute. As reported in [3], training a value function to guide search greatly improves the pass rates of θ_1 on *mathlib-valid* (see [3] for an ablation of the value function). Interestingly, the gap between θ_0 and θ_1 on *miniF2F-valid* is not as significant, demonstrating that training a value

function on proofs sampled from *mathlib-train* has limited transfer to *miniF2F-valid*. The main differences with [2], potentially explaining the gap on *minif2f-valid* (27.6% vs 23.9%), consists in the new pre-training described in section subsubsection 4.0.4.2 as well as the use of a more recent *mathlib* checkpoint for the `mix1`, `mix2` and `tactic` datasets.

4.0.4.5 Iterated sampling and training

Our expert iteration process takes as input:

- A set of formal statements St.
- A function $a : St \to \mathbb{N}$ indicating the number of proof search attempts to run per statement at each iteration.
- A base model θ_0 to fine-tune from at each iteration.
- A *mathlib* bootstrapped model θ_1 trained on both objectives.

Each iteration k consists in sampling proof searches for statements in St using θ_k, filtering successful proof searches S_k to extract a new dataset D_k, and fine-tuning θ_0 on it to obtain θ_{k+1}, on which we can iterate. To sample proof searches from St we use the best-first search described in [3] with the value function described in section subsubsection 4.0.4.3. We attempt $a(s \in St)$ proof searches for each statement s with $d = 512$ expansions and $e = 8$ samples per expansion. We denote the set of successful proof searches for iteration k as S_k.

Using S_k we generate datasets D_k by concatenating:

- The initial `tactic` dataset (*proofstep objective*)
- A deduplicated set of proofsteps extracted from the proofs in $\bigcup_{1 \leq i \leq k} S_k$ (*proofstep objective*)
- A deduplicated set of proofsize tuples (goals and proofsize) extracted from the full proof searches in $\bigcup_{1 \leq i \leq k} S_k$ (*proofsize objective*)

Note that we use a global deduplication across iterations for both proofsteps and proofsize tuples which we found to be important to maintain the stability of the expert iteration procedure. This global deduplication is somewhat equivalent for each statement to growing a unique proof tree by aggregating all the proof searches that have been run for it across

iterations. This virtual proof tree accumulates a growing number of positive proof paths as well as a growing number of visited goals that remain unproven. We use these goals as negative examples for the *proofsize objective*, labeling them with an infinite proofsize. Positive goals are deduplicated keeping the minimum proof sizes across proof searches.

Finally θ_k is obtained by fine-tuning θ_0 for exactly one epoch on D_k. Note that the initial `tactic` dataset is included in each D_k, despite θ_0 being already trained on it (along with `mix1`). We found this repetition to be beneficial overall (as it adds the *mathlib* extracted proofsteps to our deduplicated per statements virtual proof trees) despite it leading to a slight overfit on the `tactic` dataset in terms of validation loss.

4.0.4.6 Expert iteration on *mathlib-train*

In this section we propose to set *St* to the statements in *mathlib-train*, run our expert iteration process with it and report performance on both *mathlib-valid* and *miniF2F-valid*. Performance is reported in terms of pass rate (percentage of successful proof searches) as a function of the number of attempts per statement, noted *pass@k* where k is the number of attempts per statement at test time. To reduce noise in these metrics we generally run more than k attempts at test time (generally 32 to compute *pass*@1 and *pass*@8), averaging across attempts as needed to obtain a smoother *pass@k* value.

Given the large number of statements in *mathlib-train* (25k) we uniformly set $a = 1$ and use θ_0 and θ_1 as described in section subsubsection 4.0.4.4 and report *pass@1* and *pass@8* across 8 iterations in Figure 6. The *pass@1* on *mathlib-valid* goes from 56.3% for θ_1 to 62.6% for θ_9. The performance steadily improves and follows a clear logarithmic scaling law on *mathlib-valid*. It is also notable that, initially, transfer to out-of-distribution *minif2f-valid* appears limited but eventually kicks in as we reach better performance on *mathlib-valid*. This demonstrates that the expert iteration process does not just overfit to *mathlib* but also leads to improved performance on out-of-distribution statements.

We define the cumulative pass rate at iteration k as the pass rate consisting of all proof searches up to iteration k (necessarily monotonic in k). Since we set $a = 16$ for evaluation on *mathlib-valid* and *minif2f-valid* at each iteration, the cumulative pass rate at iteration

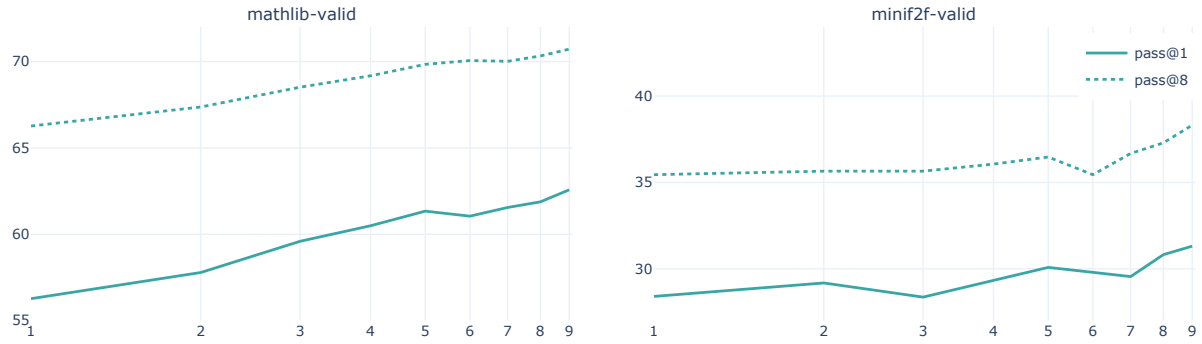

Figure 6: *pass@1* (plain) and *pass@8* (dotted) for *mathlib-valid* and *minif2f-valid* when running eight expert iterations with *St* set to be the statements in *mathlib-train*. The x-axis is log-scaled. It corresponds to the indices of the θ_k models and serves as a good proxy to compute (the amount of test-time and train-time compute per iteration being fixed). The y-axis is scaled linearly and simply shifted between the two graphs (spans an equal range).

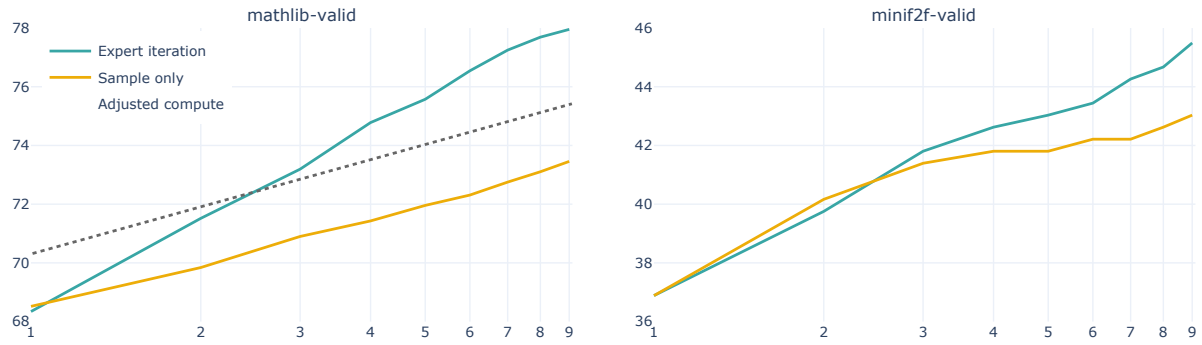

Figure 7: Cumulative pass rate for our *expert iteration* loop as well as a *sample only* loop where we skip re-training the model between iterations. The *adjusted compute* line is computed by fitting the *sample only* curve and shifting it to approximate a setup where we would focus all the additional compute used by expert iteration (sampling training data from *mathlib-train* as well as re-training models at each iteration) towards running proof searches against *mathlib-valid*.

k can be seen as a noisy ensembled *pass@16k* (multiple models (θ_k), no averaging). In Figure 7, we report this cumulative pass rate for two iteration loops, our normal one and a sampling-only loop where we skip re-training the model between iterations and solely sample from θ_1. This directly compares test-time compute scaling (scaling proof search attempts) to expert iteration scaling (interleaved training on new data sampled from *mathlib-train*) and provides a very clear visualization of the gains of expert iteration. For a fair comparison, we also report an *adjusted compute* line which approximates the test-time performance we would get at each iteration if we were to focus all the additional compute used by expert iteration (sampling proofs from *mathlib-train* as well as re-training models at each iteration) towards solely running proof searches against *mathlib-valid*.

As shown by Figure 7, the scaling exponent of expert iteration is substantially higher than the scaling exponent associated with solely scaling test-time compute (running more proof searches), demonstrating the clear benefit of expert iteration. We'll denote the fully iterated model from this section as $\theta_9^{mathlib}$.

Even in the presence of ground-truth proofs for each of the statements in *mathlib-train* (`tactic` dataset), expert iteration generates data that further improves the performance of the model. The number of statements proved in *mathlib-train* goes from 17390 (67.8%) at iteration 1 to 19476 (76.0%) at iteration 9, while the average proof length of these statements goes from 4.8 to 4.0. We hypothesize that this continuously improving performance through expert iteration stems from two effects: (1) the model finding new original proofs for the same statements and (2) the model closing marginally harder statements at each iteration – which in turn provides more useful training data for the next iteration. By iteration 9, the model is trained on more than 90% generated data. We present in Appendix subsection A.5.6 a few examples of original proofs found by our models on *mathlib-train* compared with their ground-truth versions.

To verify our hypothesis that expert iteration is capable of closing a curriculum of increasingly difficult problems out of a set of problem statements, and that this capability is independent of having access to ground-truth proofs, we propose in the next section to study expert iteration applied to a synthetic dataset of statements, without ground-truth proofs, and for which we have fine-grained control on the difficulty of each statement.

4.0.5 Statement curriculum learning

In this section we focus on running expert iteration on synthetic statements generated by an inequality generator. The use of synthetic statements enables us to control the difficulty of each statement to present evidence that expert iteration can hill-climb the intrinsic difficulty gradient of the resulting set of statements. In particular, we show that, at fixed compute budget, expert iteration eventually closes proofs of hard statements that remain completely out of reach of simply sampling proof searches without interleaved training.

4.0.5.1 Synthetic inequality generator

We designed a synthetic inequality statement generator for Lean in the spirit of the INT [111] generator. We give concrete examples and more details in subsubsection A.5.2.2. The generator consists of three phases:

The first phase consists in generating seed expressions for which we track the sign. We start by initializing an expression set E composed of tuples of expressions and sign constraints, by generating n_v variable names (letters) assumed strictly positive as well as n_n integers (for which we know the sign). For N_S rounds, we compose elements of E using unary $(log(\cdot), log(1/\cdot), sqrt(\cdot))$ or binary operations $(+, -, \times, /, \wedge, max, min)$ for which we can deduce the sign based on the sign condition of the input expression(s) and re-inject the resulting expression and sign constraint in E. This produces a set E of signed seed expressions of size $n_v + n_n + N_S$.

The second phase consists in generating inequalities from well known inequality theorems (AM-GM, Trivial inequality, Cauchy-Schwarz, Bernoulli, Young, Hölder) taking as input to these theorems expressions from E based on the sign constraints required for each theorem. We finally compose these inequalities N_D times using the theorems detailed in Appendix subsection A.5.2. The resulting inequality is a composed inequality of depth N_D based on $n_v + n_n + N_S$ seed expressions.

We finally post-process these inequalities so that they are parsable by Lean and run them through Lean's `simp` tactic for a final simplification.

N_D and N_S together control for the difficulty of the resulting inequality. N_D controls

depth of composition, while N_S controls for obfuscation as it increases the complexity of the input expressions to the composed inequalities. We report in Appendix subsection A.5.2 examples of generated inequalities for various values of N_D and N_S.

Using this generator we generate a curriculum of 5600 inequality statements (for which we don't have proofs), 100 for each values of $0 \leq N_S \leq 7$ and $0 \leq N_D \leq 6$ (taking $n_n = 4$ and randomly sampling $2 \leq n_v \leq 8$ at each generation). We denote this set of statements as *synth-ineq*.

To bootstrap our models capabilities on this specific task, we also generate 100 statements of low difficulty ($N_D = 1$ and $N_S = 5$) and formalize a proof for each of these statements. We refer to this dataset as *synth-ineq-train*. In the rest of this section we adjunct this training dataset to the `tactic` dataset used to train our models.

4.0.5.2 Expert iteration on synthetic inequality statements

In this section we propose to set *St* to the union of the statements in *mathlib-train* and *synth-ineq*. Again, we uniformly set $a = 1$ and use θ_0 and θ_1 as described in section subsubsection 4.0.4.4, except that they are now also trained on *synth-ineq-train*.

Similarly to the previous section, we report in Figure 8 the cumulative pass rate for two loops, our standard expert iteration loop, and a proof search only loop where we don't interleave training between iterations. The pass rates are reported split by values of N_D (pooling together $0 \leq N_S \leq 7$) which we found to be the main driver for difficulty.

Despite the challenging nature of these synthetic inequalities, Figure 8 demonstrates that expert iteration is capable of climbing the intrinsic curriculum induced by *synth-ineq*. In particular, expert iteration is capable of closing 6 problems of difficulty $N_D = 6$ without having been provided with any seed ground-truth proof for this difficulty level. Note that difficulty $N_D = 6$ remains completely out of reach of simply scaling the number of attempts per statements (the *sample only* loop remaining stuck at 0 for $N_D = 6$).

This confirms on our synthetic statements dataset *synth-ineq* that not only expert iteration is capable of climbing the curricula occurring in a set of statements, but this process also enables the emergence of new capabilities without the need for ground-truth proofs (ability

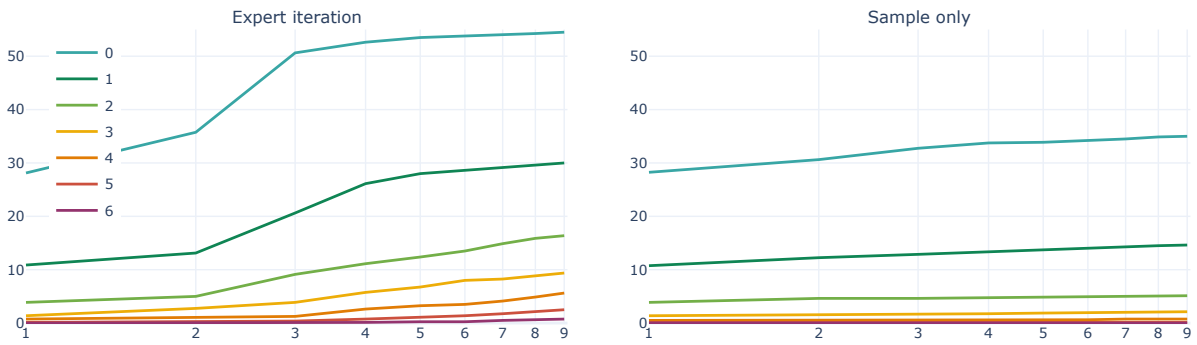

Figure 8: Cumulative pass rate for our *expert iteration* loop as well as a *sample only* loop where we skip re-training the model between iterations. Pass rates are reported for each value of N_D (pooling together $0 \leq N_S \leq 7$).

to close, highly challenging, deeply composed inequalities).

4.0.6 Targeting *miniF2F*

Motivated by the results from Section subsection 4.0.5, we curated and manually formalized a set of math exercises to target *miniF2F*. *miniF2F* statements being quite out of distribution compared to the distribution of statements present in *mathlib* (which typically includes generic theorems and lemmas but very few exercises), we hypothesized that if the difficulty of this set of statements was made varied enough, expert iteration could potentially leverage it to effectively shift our models' distribution closer to *miniF2F*'s, and in turn, improve their eventual performance on it.

4.0.6.1 Formalization effort

We manually formalized 327 statements drawn from the following sources:

- **AOPS Books [112, 113]**: 302 examples and exercises. The books are classic problem solving textbooks for students in grades 7-12 preparing for contests such as AMCs and

AIMEs. We skipped problems that were too challenging to formalize due to missing infrastructure in *mathlib* or non-suitable format for formalization (see section *Formalization effort and challenges* in [2]).

- **MATH [114] dataset**: 25 problems. All problems were drawn from the train split of the dataset, focusing on difficulty-5 problems (*miniF2F* only contains problems from the test split).

We refer to [2] for more details on the formalization procedure and the typical time needed for it as these problems were formalized in similar conditions. We denote this set of statements as *miniF2F-curriculum* and verified (based on problem provenance and manual inspection of statements) that it had an empty intersection with *miniF2F-{test,valid}*.

4.0.6.2 Transfer to *miniF2F*

In this section we propose to set St to the union of the statements in *mathlib-train*, *synth-ineq* and *miniF2F-curriculum*. We uniformly set $a = 1$ on *mathlib-train* and *synth-ineq* and $a = 8$ on *miniF2F-curriculum* and use θ_0 and θ_1 as described in section subsection 4.0.5.

Similarly to previous sections, we report in Figure 9 (left) the cumulative pass rate on *miniF2F-valid* of our full curriculum expert iteration loop and compare them with the *mathlib-train* only expert iteration from section subsubsection 4.0.4.6. Since more compute is deployed in our full-curriculum loop (more statements) we also report a *mathlib-train* only loop taking $a = 2$. At the end of the expert iteration, 100 out of the 327 statements from *miniF2F-curriculum* end up being closed, suggesting a lack of density in our manually formalized set of statement.

We also report in Figure 9 (right) the *pass@1* and *pass@8* for our full curriculum expert iteration loop. The steady improvement on *miniF2F-valid* shows that the expert iteration procedure we propose does not overfit on the statements that compose the curriculum it uses. Despite the potential inefficiency of our curriculum, the improved performance associated with its use demonstrates, as hypothesized, an effective transfer between *miniF2F-curriculum, synth-ineq* and *miniF2F-valid* through expert iteration. We'll denote the fully iterated model from this section as θ_9^{full}.

Figure 9: **Top:** cumulative pass rate on *miniF2F-valid* for our expert iteration loop using our full curriculum (*mathlib-train*, *synth-ineq* and *miniF2F-curriculum*) compared to the expert iteration loop from section subsubsection 4.0.4.6. The total number of attempts per iteration in our *full* loop is $25k + 5.6k + 8 * 327 \approx 33.2k$, which means the total compute deployed is higher than in the *mathlib-train* only loop ($25k$). We therefore also report in dotted a *mathlib-train* only loop, taking $a = 2$, whose total number of attempts per iteration is $\approx 50k$. **Bottom:** *pass@1* (plain) and *pass@8* (dotted) for our expert iteration loop using our full curriculum (*mathlib-train*, *synth-ineq* and *miniF2F-curriculum*) compared to the *expert iteration* loop from section subsubsection 4.0.4.6.

4.0.6.3 Results

We report in table Table 5 the pass rates on *mathlib-{valid, test}* and *miniF2F-{valid, test}* for the models trained in previous sections, namely θ_1, $\theta_9^{mathlib}$, and θ_9^{full}. We achieve a 47.3% pass rate (using $a = 64$ attempts) on *miniF2F-valid* and a 36.6% pass rate on *miniF2F-test*, substantially improving from the previous state-of-the-art [2].

These results include the resolution of twenty-six AMC12[4] problems, six AIME[5] problems and two problems adapted from the IMOs[6]. Out of these statements, four AMC12 problems (`amc12b_2020_p5`, `amc12a_2009_p9`, `amc12a_2003_p24`, `amc12b_2003_p17`), two AIME problems (`aime_1984_p1`, `aime_1990_p4`), and two IMO-adapted problems (`imo_1961_p1`, `imo_1964_p2`) are uniquely solved by expert iterated models, the two IMO-adapted and the two AIME problems being uniquely solved by θ_9^{full}. We provide a selection of the proofs found by our models for these statements in Appendix subsection A.5.10.

Note that one of the IMO-adapted problems solved from *miniF2F-valid*, `imo_1961_p1`[7], is a much weaker version of the informal problem[8] as it provides the conditions asked to be determined by the original question and requires instead a proof of only one side of the original equivalence (see section *Formalization effort and challenges* in [2]).

Also, we achieve a higher than 75% pass rate (using $a = 64$ attempts) on *mathlib-{valid, test}* (a new state-of-the-art as well) suggesting that our models have the potential to be effectively leveraged as proof assistants in the formalization efforts associated with *mathlib*.

4.0.7 Discussion

4.0.7.1 Model Size

Throughout this section, we used a single model size (774m trainable parameters). We briefly experimented with different model sizes (not reported in this section) and found that model size scaling is not as straightforward as in the case of unsupervised learning [24]. We

[4] `https://www.maa.org/math-competitions/amc-1012`
[5] `https://www.maa.org/math-competitions/invitational-competitions`
[6] `https://www.imo-official.org/`
[7] `https://github.com/openai/miniF2F/blob/v1/lean/src/valid.lean`
[8] `https://artofproblemsolving.com/wiki/index.php/1961_IMO_Problems/Problem_1`

Model	*pass@1*	*pass@8*	*pass@64*
mathlib-valid			
PACT [1]	48.4%	-	-
θ_1	56.3%	66.3%	72.0%
$\theta_9^{mathlib}$	**62.6%**	**70.7%**	**75.8%**
θ_9^{full}	61.7%	69.8%	75.3%
mathlib-test			
θ_1	56.5%	66.9%	73.7%
$\theta_9^{mathlib}$	**63.0%**	71.5%	**77.1%**
θ_9^{full}	62.9%	**71.6%**	76.3%
miniF2F-valid			
PACT [2]	23.9%	29.3%	-
θ_1	28.5%	35.5%	41.2%
$\theta_9^{mathlib}$	31.3%	38.3%	44.1%
θ_9^{full}	**33.6%**	**41.2%**	**47.3%**
miniF2F-test			
PACT [2]	24.6%	29.2%	-
θ_1	25.9%	31.1%	33.6%
$\theta_9^{mathlib}$	27.2%	33.0%	35.2%
θ_9^{full}	**29.6%**	**34.5%**	**36.6%**

Table 5: Performance of θ_1 (value-function based search), $\theta_9^{mathlib}$ (expert iterated on *mathlib-train*) and θ_9^{full} (expert iterated on our full curriculum) on *mathlib-{valid, test}* and *miniF2F-{valid, test}*. All proof searches are run with $d = 512$ and $e = 8$.

found that bigger models are better, in the sense that they consistently exhibit higher *pass@1*. But, they are also much more expensive to sample from. And despite their *pass@1* being

higher, it is often the case that for a fixed amount of compute, sampling more attempts from a smaller model leads to a better final performance.

For the compute budget we had available, we estimated the model size we used to be a compelling trade-off. We leave as future work a more thorough study of these dynamics to better understand the different compute frontiers involved. Indicatively, with our 774m parameters model, running a full expert iteration to train θ_9^{full} required about 2000 A100 days of compute. Running one full proof search ($a = 1$ $d = 512$ $e = 8$) when properly parallelized, requires on average about 0.1 A100 hour of compute.

4.0.7.2 Qualitative analysis of proofs

In this section we provide qualitative insights in the nature of the proofs found by our models, which we believe are useful to build a better intuition of their capabilities beyond pass rate numbers. Throughout this section, we refer to statements and solutions found by our models that are presented in Appendix subsection A.5.10 along with comments describing the specificity of each proof.

First, we observe that a large number of olympiad problems that are designed to be computationally challenging for humans are rendered trivial for our models through the use of Lean tactics. As an example, `mathd_numbertheory_447` which is not necessarily considered straightforward for humans, can be closed in Lean by a simple `refl` (proof found by our models).

In recent years, Lean's `mathlib` community has developed high-powered tactics such as `linarith/nlinarith` (solves (non)linear inequalities), `norm_num` (normalizes numerical expressions), `simp` (simplifies goals and hypotheses) and `ring` (normalizes expressions in a ring). These tactics can be used with arguments to guide their underlying search procedure. As mentioned in [2], we confirm here that our models acquire advanced capabilities to leverage these high-level tactics by providing exogenous arguments which are not present in the current tactic state. The generation of these exogenous arguments through language modeling seems to require a non-trivial amount of mathematical intuition. The problems `imo_1964_p2`, `imo_1961_p1` and `aime_1990_p15` are good examples of such uses.

We have also observed a number of proofs that require multiple non-trivial reasoning steps through the use of lower-level tactics such as `use`, `have`, or `by_cases` that generally involve producing a witness or chaining implications, requiring the generation of context specific exogenous terms. These interesting reasoning steps are structurally different from simple normalization, simplification and rewriting of hypotheses or goals because they heavily rely on our models ability to generate meaningful cuts or witnesses. This capability is, in our opinion, the most exciting stepping stone towards solving more challenging mathematical problems. See, `aopsbook_v2_c8_ex1`, `amc12b_2020_p6` and `mathd_train_algebra_217` for examples of such proofs.

More generally, we also observe that proofs generated by our models have a distinctive style compared to proofs formalized by humans. This stems in part from the model's capability to leverage high-level tactics in a way that is challenging for humans as discussed in this section (e.g. one-liners such as `nlinarith [sq_nonneg (x - y), sq_nonneg (y - z)]` where humans would generally decompose the problem in a less machine-like way). Additionally, as a result of our search procedure and despite the bias towards shorter proofs introduced by our value function, extraneous proofsteps (such as reversion/introduction of hypotheses, or no-op rewrites) are often interleaved with useful ones, which rarely happens in human formalizations.

4.0.7.3 Limitations

Despite our models' capability, as discussed in section subsubsection 4.0.7.2, to generate cuts and witnesses, we believe that their current main limitation lies in their inability (under our proposed search procedure) to chain more than two or three non-trivial steps of mathematical reasoning, preventing them from consistently (instead of exceptionally) solving challenging olympiad problems. We've been repeatedly impressed by the complexity of some of the proofsteps generated by our models. But, proofs requiring many of such reasoning steps remain beyond our current compute horizon. Even if we solved a few challenging olympiad problems, which is in itself very encouraging, we are still very far from being competitive with the brightest students in these competitions.

While our models have demonstrated some capabilities to generate cuts, the cuts they generate are often shallow (they involve only a few proofsteps and don't necessarily deeply change the structure of the proof). We believe that studying language models' ability to generate cuts, and designing search procedures that leverage that capability (related ideas can be found in [115]), are interesting avenues of research to alleviate this limitation.

5.0 Conclusions

In section 3.1 we presented PACT, a method for extracting additional training data for a language-model based neural theorem prover from low-level proof artifacts. There is a sense in which PACT is merely an application of the well known principle that compute in the form of search should be exchanged for training signal whenever possible. In Lean, typeclass inference relies on a backtracking Prolog-style search; the elaborator performs search to disambiguate overloaded notation and infer types; Lean tactics have complex semantics precisely because they can perform search to find subproofs automatically. The work done by these subroutines is preserved in the proof artifacts, and PACT can be viewed as a way of extracting this information offline for more training signal.

We have presented PACT as one way of addressing the data scarcity issue for learning theorem proving from human tactic scripts in proof assistant libraries. Another well-studied solution for this is expert iteration and reinforcement learning. In the setting of HOL Light, and under the assumption of a hardcoded finite action space of tactics, [40] in conjunction with supervised seed data was able to achieve up to 70% proof success rate on the HOList theorem proving task. Similarly, in a set-up much closer to ours, MM GPT-f [48] demonstrated the feasibility of expert iteration when using generative language models for theorem proving.

Within a fixed corpus of theorems (and hence proof terms), PACT and RL are fundamentally constrained by a lack of exploration. As the performance of the theorem proving agent improves, it will eventually saturate and become starved for data, at which point its curriculum will need to be expanded. Although self-supervised methods such as PACT represent a way to significantly improve the data-efficiency of reinforcement learning loops over existing theorem prover libraries, the development of continuously self-improving and infinitely scalable neural theorem provers remains contingent on sufficiently powerful exploration and automated curriculum generation.

We can view the other contributions in this dissertation as steps in those directions. In section 3.2, we proposed a new method, SPT-Aug, for generating novel training examples for language models in the formal theorem proving domain by synthesizing entirely new

proof terms, and using Lean's kernel to filter for the typecorrect terms and extract their type (i.e. the theorem being proved). This method is shown to improve the perplexity of both conditional and unconditional proof term language modeling on a held-out test set. The result also withstands an ablation of the regularizing effect of simply mixing in a large quantity of related data. We see SPT-Aug as a promising direction for further research in this domain. We remark that that the unconditional version of SPT-Aug gives a method for sampling new theorem statements alongside the proofs, an ingredient which is missing from both PACT (section 3.1 and statement curriculum learning, which operates against a fixed curriculum of theorems (chapter 4)). Future elaborations of this method could focus on incorporating SPT-Aug into a neural theorem proving loop to improve performance on theorem proving rather than just perplexity scores on held-out data. Another immediate extension of SPT-Aug applicable to the methods in chapter 4 would to sample *types* instead and use these in conjunction with some kind of filtering to seed an expert iteration loop for statement curriculum learning.

We presented an expert iteration procedure for *GPT-f* [3] called *statement curriculum learning* in chapter 4. This procedure is capable of solving a curriculum of increasingly difficult problems out of a set of formal statements of sufficiently varied difficulty. We achieved new state-of-the-art results on the *miniF2F* benchmark (derived from math Olympiad problems) and discussed proofs generated by our models, including proofs for multiple challenging olympiad problems. Our results suggest that the lack of self-play in the formal mathematics setup can be effectively compensated for by automatically as well as manually curated sets of formal statements, which are much cheaper to formalize than full proofs. Finally, we hope that the *statement curriculum learning* methodology we presented in this work will help accelerate progress in automated reasoning, especially if scaled with future progress in automated generation and curation of formal statements.

A.1 Additional materials for section 3.1

A.1.1 Additional background for PACT

Lean's fundamental logic is a dependent type theory called the calculus of inductive constructions [60]. This design means that terms (4, $x+y$, f), types ($\mathbb{N}$, `list` $\mathbb{Z}$, $\alpha \to \beta$) and proofs are all represented with a single datatype called an *expression*. Given an environment of available constants and definitions and a context Γ of variables, Lean can infer a type α for each well-formed expression t. A *proof term* is a Lean expression whose type is a proposition–this proof term serves as a checkable artifact for verifying the proposition. Lean uses a small, trusted kernel to verify proof terms.

Tactics in Lean are metaprograms [6], which can construct Lean expressions, such as terms. A *tactic state* which tracks the list of open goals and other metadata is threaded through each tactic invocation. Lean has special support for treating tactics as an extensible DSL; this DSL is how Lean is typically used as an interactive theorem prover. The DSL amounts to a linear chain of comma-separated invocations. The process of interactive proving is mediated through Lean's language server, which presents the context and type for the current goal in the proof to the user, dependent on where their cursor is in the source text. The *tactic prediction* task is to predict the next tactic given this goal state. We extract supervised training data for this task by extracting all human-supplied proof steps from Lean's `mathlib`.

The *tactic state* is an object threaded through each invocation of a tactic. The tactic state maintains, among other things, a *context* of metavariables—placeholders in which expressions will later be substituted. At each point of the proof, one or more of these metavariables are selected as the *goal* of the current tactic state. As the proof progresses, there are multiple values to be found.

Consider this (modified) example of a tactic proof from the library.

```
theorem int.sub_ne_zero_of_ne : ∀ (a b : ℤ), a ≠ b -> a - b ≠ 0 :=
begin
   intros a b h hab,
   apply h,
   apply int.eq_of_sub_eq_zero hab,
```

```
end
```

Each tactic line modifies the proof state, which we explicitly annotate below with comments between each tactic.

```
theorem int.sub_ne_zero_of_ne : ∀ (a b : ℤ), a ≠ b -> a - b ≠ 0 :=
begin
  -- ⊢ ∀ (a b : ℤ), a ≠ b → a - b ≠ 0
  intros a b h hab,
  -- a b : ℤ,
  -- h : a ≠ b,
  -- hab : a - b = 0
  -- ⊢ false
  apply h,
  -- a b : ℤ,
  -- h : a ≠ b,
  -- hab : a - b = 0
  -- ⊢ a = b
  apply int.eq_of_sub_eq_zero hab,
  -- no goals
end
```

Our proofstep objective is to predict the tactic applied to a given tactic state.

Lean stores this proof internally as a proof term:

```
theorem int.sub_ne_zero_of_ne : ∀ (a b : ℤ), a ≠ b → a - b ≠ 0 :=
λ (a b : ℤ) (h : a ≠ b), id (λ (hab : a - b = 0), h (int.eq_of_sub_eq_zero hab))
```

Since this proof term is just stored internally as a tree, any branch of this term tree can be removed, to create a hole _, for example:

```
λ (a b : ℤ) (h : a ≠ b), id (λ (hab : a - b = 0), h  _)
```

Lean will automatically provide a list of both the local context and the type of a term needed to fill that hole as shown below. Notice this is the same as a tactic state we saw from the term proof above.

```
a b : ℤ,
h : a ≠ b,
hab : a - b = 0
⊢ a = b
```

Using this methodology of following proof term trees, we can mine low level proof data for every node of a term proof to produce the PACT dataset described in Section 3.1.2.1.

A.2 Datasets for section 3.1

A.2.1 Pre-training datasets

We pre-train our models on `WebMath`, as described in [48]. All models, including both the `WebMath` pre-trained models and the models used in ablations that were not pre-trained on `WebMath`, were first pre-trained on the mix used by GPT-3 [23]. This mix includes filtered data from `CommonCrawl`, `WebText2`, `Book1`, `Book2`, and `Wikipedia`. `WebMath`'s data set includes Python-only `GitHub` data, as well as `arXiv` and `Math StackExchange`.

From these datasets, a potential risk for test-set contamination (presence of `mathlib`) exists for the crawled datasets, namely `CommonCrawl`, `WebText2`, and (in case of a filtering bug) Python-only `GitHub`. The other datasets (in particular `arXiv` and `Math StackExchange`) may contain short references of `mathlib` code but in shape and forms that would not lead to effective contamination.

To assess the contamination risk related with the crawled datasets, we searched `CommonCrawl`, `WebText2`, `arXiv`, Python-only `GitHub`, and `Math StackExchange` for test theorems. For example, given the test theorem `nat.div_eq_sub_div` we searched for any occurrences of the string `div_eq_sub_div`. Of over 3000 test theorem names, we found 595 which occurred in the datasets. Many instances were innocuous, but some were in Lean files, and in some cases there was a proof of a test theorem. There were also 160 additional test theorems with no underscore in their name, which we did not check, but whose name is likely to be found in the datasets. (There is no need to check for training theorems since they are already in the training data and it would not constitute contamination.) We re-calculated the pass-rates of the results in Figure 2 omitting these 755 test theorems. This decreases the reported pass-rates slightly, ranging from 0.6 to 1.1 percentage points. The adjusted pass-rate of our best model `WebMath > mix1 + mix2 + tactic` is 47.4%, a decrease of 1 percentage point. Our main results still hold even with the adjusted pass-rates.

We also looked at the results for 1,350 and 544 test theorems, respectively, added to Lean and `mathlib` after April 18, 2020, and September 11, 2020. These theorems were part of the originally extracted data, unlike the theorems in `future-mathlib`. The pass-rates for

the `WebMath > mix1 + mix2 + tactic` model on these restricted sets of test theorems are 45.6% and 43.3%, respectively.

We also looked for the following Metamath specific and HOL specific strings in `CommonCrawl`, `WebText2`, and Python-only `GitHub`:

```
Metamath:
    "( ph -> A = C )"
    "( ph -> A R C )"
    "( sqrt ` 2 ) e/ QQ"
HOL:
    "apply (rule "
    "apply (drule "
```

We found 0 occurrence of the Metamath-related strings but interestingly found a non-negligible amount of HOL-related documents, which does not constitute a test-set contamination but potentially benefits the downstream tasks studied in this section.

While our results show a significant benefit to pre-training on `WebMath`, it is unclear exactly how pre-training helps. Since Lean's theorem names are made of coded mathematical phases, e.g. `affine.simplex.dist_circumcenter_eq_circumradius`, it is not unreasonable to suspect that important statistical connections are extracted from math sources. It is even possible that simple instances of auto-formalization or ITP translation are happening. There is prior work [116, 117, 118] suggesting that both of these are possible. From the point of view of a `lean-gptf` end-user, any such extraction of prior, publicly available data is useful and helpful. Nonetheless, our results are of a different nature than other AI for theorem proving research which do not use data outside of a given theorem proving library. This should be taken into account in any future comparisons and benchmarks.

A.2.2 Dataset sizes

- `tactic`: ≈128K examples.
- `mix1`
  - **Next lemma prediction**: ≈2.5M examples
  - **Proof term prediction**: ≈2.9M examples
- `mix2`

- **Skip-proof**: ≈1.7M examples
- **Type-prediction**: ≈1.7M examples
- **Tactic state elaboration**: ≈346K examples
- **Proof term elaboration**: ≈1.0M examples
- **Premise classification**: ≈9.3M examples
- **Local context classification**: ≈2.0M examples
- **Theorem naming**: ≈32K examples.

A.2.3 Example datapoints

We present datapoints extracted from a toy example, namely the proof of the Peirce identity, viz.

```
lemma peirce_identity {P Q :Prop} : ((P → Q) → P) → P :=
begin
  apply or.elim (em P),
  intros h _,
  exact h,
  tauto!
end
```

From this, we can extract four `tactic` datapoints (i.e. human-generated tactic proof steps):

```
-- GOAL P Q : Prop ⊢ ((P → Q) → P) → P PROOFSTEP apply or.elim (em P)
-- GOAL P Q : Prop ⊢ P → ((P → Q) → P) → P  P Q : Prop ⊢ ¬P → ((P → Q) →
   P) → P PROOFSTEP intros h _
-- GOAL P Q : Prop, h : P, α : (P → Q) → P ⊢ P  P Q : Prop ⊢ ¬P → ((P → Q) →
   P) → P PROOFSTEP exact h
-- GOAL P Q : Prop ⊢ ¬P → ((P → Q) → P) → P PROOFSTEP tauto!
```

In contrast, we can extract dozens of raw PACT datapoints. Due to space constraints, we list a representative sample of four such datapoints, from each of which we can derive the nine self-supervised auxiliary PACT tasks studied in our present work. For example, proof term prediction is precisely predicting the `"proof_term"` given the concatenation of `"hyps"`, `"⊢"`, and the `"goal"`, skip-proof is predicting the `"proof_term"` given `"result"`, etc.

```
  DATAPOINT:
---
```

```
{ "decl_nm":"peirce_identity",
  "decl_tp":"∀ {P Q : Prop}, ((P → Q) → P) → P",
  "hyps":[["P", "Prop"], ["Q", "Prop"], ["α̈", "¬P"], ["α̈_1", "(P → Q) → P"],
   ["α̈_1", "¬(P → Q)"]],
  "hyps_mask":[true, false, false, false, false],
  "decl_premises":[["absurd", "∀ {a b : Prop}, a → ¬a → b"],
   ["absurd", "∀ {a b : Prop}, a → ¬a → b"],
   ["decidable.not_imp", "∀ {a b : Prop} [_inst_1 : decidable a], ¬(a → b) ↔ a ∧
    ¬b"],
   ["iff.mp", "∀ {a b : Prop}, (a ↔ b) → a → b"],
   ["and.dcases_on",
    "∀ {a b : Prop} {C : a ∧ b → Prop} (n : a ∧ b), (∀ (left : a) (right : b), C
   _) → C n"],
   ["decidable.not_or_of_imp", "∀ {a b : Prop} [_inst_1 : decidable a], (a → b) →
    ¬a ∨ b"],
   ["or.dcases_on",
    "∀ {a b : Prop} {C : a ∨ b → Prop} (n : a ∨ b), (∀ (h : a), C _) → (∀ (h :
   b), C _) → C n"],
   ["em", "∀ (p : Prop), p ∨ ¬p"],
   ["or.elim", "∀ {a b c : Prop}, a ∨ b → (a → c) → (b → c) → c"]],
  "decl_premises_mask":[false, false, true, false, false, false, false, false,
   false],
  "goal":"∀ {b : Prop} [_inst_1 : decidable P], ¬(P → b) ↔ P ∧ ¬b",
  "proof_term":"decidable.not_imp",
  "result":"λ {P Q : Prop}, (em P).elim (λ (h : P) (α̈ : (P → Q) → P), h) (λ (α̈
   : ¬P) (α̈_1 : (P → Q) → P), (decidable.not_or_of_imp α̈_1).dcases_on (λ (α̈_1
   : ¬(P → Q)), ((PREDICT Q (classical.prop_decidable P)).mp α̈_1).dcases_on (λ
   (α̈_1_left : P) (α̈_1_right : ¬Q), absurd α̈_1_left α̈)) (λ (α̈_1 : P), absurd α̈
   _1 α̈))",
  "next_lemma":["decidable.not_imp", "∀ {a b : Prop} [_inst_1 : decidable a], ¬(a
   → b) ↔ a ∧ ¬b"],
  "goal_is_prop":true,
  "verbose_proof_term":"@decidable.not_imp P",
  "verbose_goal":"∀ {b : Prop} [_inst_1 : decidable P], ¬(P → b) ↔ P ∧ ¬b",
  "verbose_result":"λ {P Q : Prop}, (em P).elim (λ (h : P) (α̈ : (P → Q) → P),
   h) (λ (α̈ : ¬P) (α̈_1 : (P → Q) → P), (@decidable.not_or_of_imp (P → Q) P
   (classical.prop_decidable (P → Q)) α̈_1).dcases_on (λ (α̈_1 : ¬(P → Q)),
   (@iff.mp (¬(P → Q)) (P ∧ ¬Q) (PREDICT Q (classical.prop_decidable P)) α̈
   _1).dcases_on (λ (α̈_1_left : P) (α̈_1_right : ¬Q), @absurd P P α̈_1_left α̈))
   (λ (α̈_1 : P), @absurd P P α̈_1 α̈))"}
---

DATAPOINT:
---
{ "decl_nm":"peirce_identity",
  "decl_tp":"∀ {P Q : Prop}, ((P → Q) → P) → P",
  "hyps":[["P", "Prop"], ["Q", "Prop"], ["α̈", "¬P"], ["α̈_1", "(P → Q) → P"],
```

```
    ["ă_1", "¬(P → Q)"]],
  "hyps_mask":[false, true, false, false, false],
  "decl_premises":[["absurd", "∀ {a b : Prop}, a → ¬a → b"],
   ["absurd", "∀ {a b : Prop}, a → ¬a → b"],
   ["decidable.not_imp", "∀ {a b : Prop} [_inst_1 : decidable a], ¬(a → b) ↔ a ∧
    ¬b"],
   ["iff.mp", "∀ {a b : Prop}, (a ↔ b) → a → b"],
   ["and.dcases_on",
    "∀ {a b : Prop} {C : a ∧ b → Prop} (n : a ∧ b), (∀ (left : a) (right : b), C
   _) → C n"],
   ["decidable.not_or_of_imp", "∀ {a b : Prop} [_inst_1 : decidable a], (a → b) →
    ¬a ∨ b"],
   ["or.dcases_on",
    "∀ {a b : Prop} {C : a ∨ b → Prop} (n : a ∨ b), (∀ (h : a), C _) → (∀ (h :
   b), C _) → C n"],
   ["em", "∀ (p : Prop), p ∨ ¬p"],
   ["or.elim", "∀ {a b c : Prop}, a ∨ b → (a → c) → (b → c) → c"]],
  "decl_premises_mask":[false, false, false, false, false, false, false, false,
   false],
  "goal":"Prop",
  "proof_term":"Q",
  "result":"λ {P Q : Prop}, (em P).elim (λ (h : P) (ă : (P → Q) → P), h) (λ (ă
   : ¬P) (ă_1 : (P → Q) → P), (decidable.not_or_of_imp ă_1).dcases_on (λ (ă_1
   : ¬(P → Q)), (decidable.not_imp.mp ă_1).dcases_on (λ (ă_1_left : P)
   (ă_1_right : ¬Q), absurd ă_1_left ă)) (λ (ă_1 : P), absurd ă_1 ă))",
  "next_lemma":["Q", "Prop"],
  "goal_is_prop":false,
  "verbose_proof_term":"Q",
  "verbose_goal":"Prop",
  "verbose_result":"λ {P Q : Prop}, (em P).elim (λ (h : P) (ă : (P → Q) → P),
   h) (λ (ă : ¬P) (ă_1 : (P → Q) → P), (@decidable.not_or_of_imp (P → Q) P
   (classical.prop_decidable (P → Q)) ă_1).dcases_on (λ (ă_1 : ¬(P → Q)),
   ((@decidable.not_imp P PREDICT (classical.prop_decidable P)).mp ă_1).dcases_on
   (λ (ă_1_left : P) (ă_1_right : ¬Q), @absurd P P ă_1_left ă)) (λ (ă_1 : P),
   @absurd P P ă_1 ă))"}
---

DATAPOINT:
---
{ "decl_nm":"peirce_identity",
  "decl_tp":"∀ {P Q : Prop}, ((P → Q) → P) → P",
  "hyps":[["P", "Prop"], ["Q", "Prop"], ["ă", "¬P"], ["ă_1", "(P → Q) → P"],
   ["ă_1", "¬(P → Q)"]],
  "hyps_mask":[true, true, false, false, false],
  "decl_premises":[["absurd", "∀ {a b : Prop}, a → ¬a → b"],
   ["absurd", "∀ {a b : Prop}, a → ¬a → b"],
   ["decidable.not_imp", "∀ {a b : Prop} [_inst_1 : decidable a], ¬(a → b) ↔ a ∧
```

```
    ¬b"],
   ["iff.mp", "∀ {a b : Prop}, (a ↔ b) → a → b"],
   ["and.dcases_on",
    "∀ {a b : Prop} {C : a ∧ b → Prop} (n : a ∧ b), (∀ (left : a) (right : b), C
   _) → C n"],
   ["decidable.not_or_of_imp", "∀ {a b : Prop} [_inst_1 : decidable a], (a → b) →
    ¬a ∨ b"],
   ["or.dcases_on",
    "∀ {a b : Prop} {C : a ∨ b → Prop} (n : a ∨ b), (∀ (h : a), C _) → (∀ (h :
   b), C _) → C n"],
   ["em", "∀ (p : Prop), p ∨ ¬p"],
   ["or.elim", "∀ {a b c : Prop}, a ∨ b → (a → c) → (b → c) → c"]],
  "decl_premises_mask":[false, false, true, false, false, false, false, false,
   false],
  "goal":"∀ [_inst_1 : decidable P], ¬(P → Q) ↔ P ∧ ¬Q",
  "proof_term":"decidable.not_imp",
  "result":"λ {P Q : Prop}, (em P).elim (λ (h : P) (ă : (P → Q) → P), h) (λ (ă
   : ¬P) (ă_1 : (P → Q) → P), (decidable.not_or_of_imp ă_1).dcases_on (λ (ă_1
   : ¬(P → Q)), ((PREDICT (classical.prop_decidable P)).mp ă_1).dcases_on (λ
   (ă_1_left : P) (ă_1_right : ¬Q), absurd ă_1_left ă)) (λ (ă_1 : P), absurd ă
   _1 ă))",
  "next_lemma":["decidable.not_imp", "∀ {a b : Prop} [_inst_1 : decidable a], ¬(a
   → b) ↔ a ∧ ¬b"],
  "goal_is_prop":true,
  "verbose_proof_term":"@decidable.not_imp P Q",
  "verbose_goal":"∀ [_inst_1 : decidable P], ¬(P → Q) ↔ P ∧ ¬Q",
  "verbose_result":"λ {P Q : Prop}, (em P).elim (λ (h : P) (ă : (P → Q) → P),
   h) (λ (ă : ¬P) (ă_1 : (P → Q) → P), (@decidable.not_or_of_imp (P → Q) P
   (classical.prop_decidable (P → Q)) ă_1).dcases_on (λ (ă_1 : ¬(P → Q)),
   (@iff.mp (¬(P → Q)) (P ∧ ¬Q) (PREDICT (classical.prop_decidable P)) ă
   _1).dcases_on (λ (ă_1_left : P) (ă_1_right : ¬Q), @absurd P P ă_1_left ă))
   (λ (ă_1 : P), @absurd P P ă_1 ă))"}
---

DATAPOINT:
---
{ "decl_nm":"peirce_identity",
  "decl_tp":"∀ {P Q : Prop}, ((P → Q) → P) → P",
  "hyps":[["P", "Prop"], ["Q", "Prop"], ["ă", "¬P"], ["ă_1", "(P → Q) → P"],
   ["ă_1", "¬(P → Q)"]],
  "hyps_mask":[false, false, false, false, false],
  "decl_premises":[["absurd", "∀ {a b : Prop}, a → ¬a → b"],
   ["absurd", "∀ {a b : Prop}, a → ¬a → b"],
   ["decidable.not_imp", "∀ {a b : Prop} [_inst_1 : decidable a], ¬(a → b) ↔ a ∧
    ¬b"],
   ["iff.mp", "∀ {a b : Prop}, (a ↔ b) → a → b"],
   ["and.dcases_on",
```

```
   "∀ {a b : Prop} {C : a ∧ b → Prop} (n : a ∧ b), (∀ (left : a) (right : b), C
   _) → C n"],
   ["decidable.not_or_of_imp", "∀ {a b : Prop} [_inst_1 : decidable a], (a → b) →
    ¬a ∨ b"],
   ["or.dcases_on",
    "∀ {a b : Prop} {C : a ∨ b → Prop} (n : a ∨ b), (∀ (h : a), C _) → (∀ (h :
   b), C _) → C n"],
   ["em", "∀ (p : Prop), p ∨ ¬p"],
   ["or.elim", "∀ {a b c : Prop}, a ∨ b → (a → c) → (b → c) → c"]],
  "decl_premises_mask":[false, false, false, false, false, false, false, false,
   false],
  "goal":"Π (a : Prop), decidable a",
  "proof_term":"classical.prop_decidable",
  "result":"λ {P Q : Prop}, (em P).elim (λ (h : P) (ă : (P → Q) → P), h) (λ (ă
   : ¬P) (ă_1 : (P → Q) → P), (decidable.not_or_of_imp ă_1).dcases_on (λ (ă_1
   : ¬(P → Q)), (decidable.not_imp.mp ă_1).dcases_on (λ (ă_1_left : P)
   (ă_1_right : ¬Q), absurd ă_1_left ă)) (λ (ă_1 : P), absurd ă_1 ă))",
  "next_lemma":["classical.prop_decidable", "Π (a : Prop), decidable a"],
  "goal_is_prop":false,
  "verbose_proof_term":"classical.prop_decidable",
  "verbose_goal":"Π (a : Prop), decidable a",
  "verbose_result":"λ {P Q : Prop}, (em P).elim (λ (h : P) (ă : (P → Q) → P),
   h) (λ (ă : ¬P) (ă_1 : (P → Q) → P), (@decidable.not_or_of_imp (P → Q) P
   (PREDICT (P → Q)) ă_1).dcases_on (λ (ă_1 : ¬(P → Q)), ((@decidable.not_imp
   P Q (PREDICT P)).mp ă_1).dcases_on (λ (ă_1_left : P) (ă_1_right : ¬Q),
   @absurd P P ă_1_left ă)) (λ (ă_1 : P), @absurd P P ă_1 ă))"}
---
```


A.3 Experiments for section 3.1

A.3.1 Chained tactic prediction

Individual Lean tactics are chained together with commas. However, the Lean interactive tactic DSL also includes a number of other tactic combinators for creating composite tactics. A frequently used combinator is the infix semicolon `t; s` which will perform the tactic `t` and then apply the tactic `s` to each of the resulting subgoals produced by `t`. Our data pipeline for human tactic proof steps treats these semicolon-chained tactics as a single string for the language modeling objective. Thus, our models learn to occasionally emit multiple-step tactic predictions using semicolons. For example, `wm-to-tt-m1-m2` solved the following lemma in

category theory with a single prediction chaining four tactics in a row:

```
theorem category_theory.grothendieck.congr
  {X Y : grothendieck F} {f g : X ⟶ Y} (h : f = g) :
  f.fiber = eq_to_hom (by subst h) ≫ g.fiber :=
begin
  rcases X; rcases Y; subst h; simp
end
```

We can measure the sophistication of predicted tactics by considering the number of successful proofs on the evaluation set that have a composite form using semicolon-chaining. Appendix A.3.1 shows that training with PACT in addition to human-made tactics causes longer semicolon-chained tactics to be successfully predicted during theorem proving. This is remarkable because the semicolon idiom is specific to the tactic DSL and does not occur in the PACT data whatsoever. And yet, the co-training causes longer and more frequent successful composite tactic predictions.

Model	1;	2;	3;	4;	Mean
wm-to-tt	215	49	2	0	1.199
wm-to-tt-m1	186	39	5	1	1.225
wm-to-tt-m1-m2	**328**	**82**	**12**	**3**	**1.271**

Table 6: Counting the number of semicolon-chained tactics predicted by our models that appear *in successful proofs*. Each column headed by a number n; indicates the number of times that a suggestion appeared with n occurrences of ';'.

A.3.2 Theorem naming case study

Correct top-1 guesses	
Theorem statement	∀ {α : Type u_1} {β : Type u_2} [_inst_1 : decidable_eq α] [_inst_2 : decidable_eq β] (s : finset α) (t : finset β), s.product t = s.bUnion (λ (a : α), finset.image (λ (b : β), (a, b)) t)
Ground truth	finset.product_eq_bUnion
Theorem statement	∀ {α : Type u_1} {β : Type u_2} [_inst_1 : topological_space α] [_inst_2 : topological_space β] {f : α → β}, quotient_map f → function.surjective f
Ground truth	quotient_map.surjective
Theorem statement	∀ {α : Type u_1} {β : Type u_2} (f : α → option β) (x : option α), x.pbind (λ (a : α) (_x : a ∈ x), f a) = x.bind f
Ground truth	option.pbind_eq_bind

Figure 10: A sample of correct top-1 guesses by our best model `wm-to-tt-m1-m2` on the *theorem naming* task. We performed this experiment on the `future-mathlib` evaluation set, which comprises entirely unseen theorems added to `mathlib` only after we last extracted training data.

Incorrect guesses	
Theorem statement	∀ {α : Type u_1} (t : ordnode α) (x : α), t.dual.find_min′ x = ordnode.find_max′ x t
Guesses (top 8)	ordinal.find_min′_eq, ordinal.find_min′_eq_max′, ordinal.find_min′_def, ordinal.find_min′_eq_max, ordinal.find_min′, ordinal.dual_find_min′, ordinal.find_min′_gt, ordinal.find_min′_q
Ground truth	ordnode.find_min′_dual
Theorem statement	∀ {α : Type u_1} {β : Type u_2} {γ : Type u_3} {f : filter α} {h : set α → set β} {m : γ → β} {l : filter γ}, filter.tendsto m l (f.lift′ h) ↔ ∀ (s : set α), s ∈ f → (∀ᶠ (a : γ) in l, m a ∈ h s)
Guesses (top 8)	filter.tendsto_lift′_iff, filter.tendsto_lift′_def
Ground truth	filter.tendsto_lift′

Figure 11: A sample of incorrect guesses by our best model `wm-to-tt-m1-m2` on the *theorem naming* task. We performed this experiment on the `future-mathlib` evaluation set, which comprises entirely unseen theorems added to `mathlib` only after we last extracted training data. Most of the top-8 guesses displayed in the above table are very similar to the ground truth, in some cases being equivalent up to permutation of underscore-separated tokens. Note that for the first example, the concept of `ordnode` was not in the training data whatsoever and all predictions are in the syntactically similar `ordinal` namespace.

We included *theorem naming* as part of the PACT task suite. By `mathlib` convention, theorem names are essentially snake-cased, natural language summaries of the type signature of a theorem, and so the theorem naming task is analogous to a formal-to-informal translation task. We evaluate the ability of our best model (in terms of theorem proving success rate) `wm-to-tt-m1-m2` on its ability to guess theorem names on the completely unseen `future-mathlib` set of theorems. The distribution shift inherent in the `future-mathlib` dataset particularly impacts the theorem naming task, because many of the ground-truth names will involve names for concepts that were only defined in `mathlib` *after* we extracted our training data.

On the approximately 2.8K `future-mathlib` theorems, we queried `wm-to-tt-m1-m2` for up to $N = 16$ candidates. We order these candidates into a list `xs` by decreasing cumulative log-probability and calculate the top-K accuracy by checking if any of the first K candidates of `xs` match the ground truth exactly. The model `wm-to-tt-m1-m2` was able to achieve 20.1% top-1 accuracy, 21.1% top-3 accuracy, 26.7% top-10 accuracy, and 30.0% top-16 accuracy. We display a sample of correct top-1 guesses (Figure 10) and a sample of failed guesses in (Figure 11). We note that the failed guesses, while containing no syntactic matches, are both semantically reasonable and syntactically very similar to the ground truth.

A.3.3 Test set evaluation breakdown by module

Lean's `mathlib` is organized into top-level modules, which roughly organize theorems into mathematical subject area. In Figure 12, we break down the evaluation results on our `test` set between our PACT-trained models `wm-to-tt-m1-m2` and `wm-to-tt-m1` and our baselines `wm-to-tt` and `tidy`. We see that full PACT mostly dominates over co-training on just the `mix1` tasks over all subject areas, and that `wm-to-tt-m1` dominates the model `wm-to-tt` trained on human tactic proof steps only.

A.3.4 Baseline description

The `tidy` backend is determined by a constant oracle

```
Ω : tactic_state → list (string × float)
```

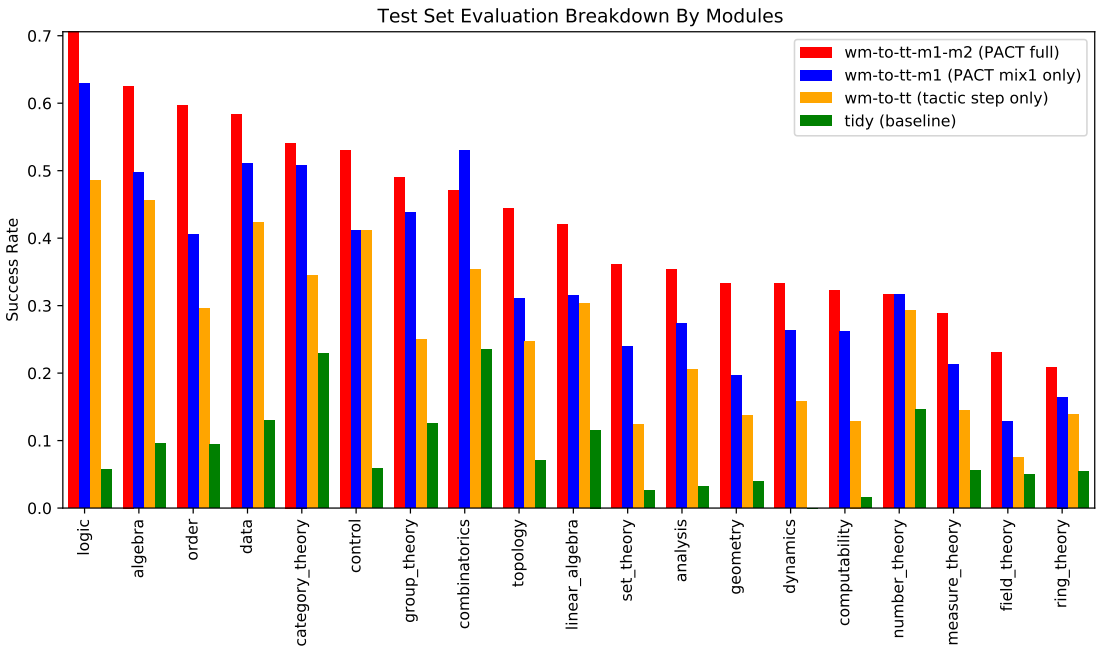

Figure 12: A breakdown of theorem proving success rate on the `test` set for `wm-to-tt-m1-m2`, `wm-to-tt-m1`, `wm-to-tt`, and the `tidy` baseline across top-level modules in Lean's `mathlib`. We see that `wm-to-tt-m1-m2` mostly dominates `wm-to-tt-m1` and the models trained using PACT dominate the model `wm-to-tt` trained on human tactic proof steps.

which always returns the same list of tactics, namely:

```
  meta def tidy_default_tactics : list (string × float) :=
  list.map (flip prod.mk 0.0) [
     "refl"
  ,  "exact dec_trivial"
  ,  "assumption"
  ,  "tactic.intros1"
  ,  "tactic.auto_cases"
  ,  "apply_auto_param"
  ,  "dsimp at *"
  ,  "simp at *"
  ,  "ext1"
  ,  "fsplit"
  ,  "injections_and_clear"
  ,  "solve_by_elim"
  ,  "norm_cast"
]
```

Unlike the `gptf` backend, which generates a list of candidates in parallel independently, `tidy` enjoys the advantage that the list of tactics it emits is carefully chosen and ordered in order to optimize the proof search—this is based on the "waterfall" technique of the human-style automated theorem prover described in [119].

A.3.5 Computational resource estimates

For each evaluation loop over the `test` set, we distributed the theorems over a pool of 32 CPU workers whose inference requests were load-balanced over 4 `V100` GPUs. Each evaluation required ≈10 hours with ≈30% GPU utilization. We observed that our evaluation was bottlenecked by inference and in practice, we hosted up to three evaluation loops at once on a VM with 80 logical cores without achieving full CPU utilization. In addition to the wall-clock timeout of 600s, we also limited the proof search to a logical timeout of `512` iterations, where one iteration corresponds to a single expansion of a node of the BFS search tree. In practice, so much time was spent either blocked on inference or performing the tactic executions in the inner loop of each iteration that we rarely exceeded the logical timeout, usually exceeding the wall-clock timeout instead.

Fine-tuning on our largest dataset `mix1 + mix2 + tactic` required 26 hours using 64 `A100` GPUs exhibiting high `FP16` usage, totalling an estimated ≈1.5K `A100(FP16)`-hours.

This gives an estimated cost of 17.33 A100(FP16)-hours per billion elapsed tokens during training. We note that when calculating the number of elapsed tokens for training, we overestimate the actual number of tokens effectively trained on by summing full context windows (in this case, 2048 tokens).

A.4 Example proofs for section 3.1

Lean's mathlib is one of the most active open-source software projects in the world. More than one-third of the proofs found by our models are shorter and produce smaller proof terms than the ground truth, leading to dozens of GPT-f co-authored commits to mathlib. We examine some of the proofs found by our models in more detail.

A.4.1 lie_algebra.morphism.map_bot_iff

This proof produces a proof term which is 4X smaller than the original:

```
lemma map_bot_iff : I.map f = ⊥ ↔ I ≤ f.ker :=
by { rw ← le_bot_iff, apply lie_ideal.map_le_iff_le_comap }
```

The original, human-written proof is much longer, viz.

```
lemma map_bot_iff : I.map f = ⊥ ↔ I ≤ f.ker :=
begin
  rw le_ker_iff, unfold lie_ideal.map, split; intros h,
  { rwa [eq_bot_iff, lie_submodule.lie_span_le, set.image_subset_iff,
    lie_submodule.bot_coe] at h,},
  { suffices : f '' I = ↑(⊥ : lie_ideal R L'), { rw [this,
    lie_submodule.lie_span_eq], },
    ext x, rw [lie_submodule.bot_coe, set.mem_singleton_iff, set.mem_image],
    split,
    { rintros ⟨y, hy, hx⟩, rw ← hx, exact h y hy, },
    { intros hx, use 0, simp [hx], }, },
end
```

A.4.2 primrec.of_equiv

This proof produces a proof term which is 12X smaller than the original:

```
theorem of_equiv {β} {e : β ≃ α} :
  by haveI := primcodable.of_equiv α e; exact
  primrec e :=
by letI : primcodable β := primcodable.of_equiv α e; exact encode_iff.1
    primrec.encode
```

The author of the original proof and maintainer of that package commented:

> `encode_iff.1 primrec.encode` is clever, it's a way to translate `primrec` across an equivalence when the encode function is defined as `encode x = encode (e x)` where `e` is the isomorphism.

As far as they knew, this trick was never used before in the `computability` package.

A.4.3 `real.tan_eq_sin_div_cos`

This proof demonstrates our model's library knowledge and ability at premise selection. This lemma takes a trigonometric identify over the complex numbers and translates it to the corresponding identity over the reals.

```
lemma real.tan_eq_sin_div_cos (x : ℝ) : tan x = sin x / cos x :=
begin
  rw ← of_real_inj,
  simp only [complex.tan_eq_sin_div_cos, of_real_sin, of_real_cos, of_real_div,
    of_real_tan]
end
```

Our model was able to predict this entire list of `simp` lemmas in one shot. Note that the lemma `complex.tan_eq_sin_div_cos` in this list is the *complex number* version of the result, i.e. `∀ (x : ℂ), tan x = sin x / cos x`. The previous human-written version of the proof did not use the more general version of the lemma on complex numbers, demonstrating our model's ability to find more general cases of lemmas. We contrast this with the human-written ground truth, which is more complex and performs a case analysis using the complex cosine:

```
  lemma tan_eq_sin_div_cos : tan x = sin x / cos x :=
if h : complex.cos x = 0 then by simp [sin, cos, tan, *, complex.tan,
    div_eq_mul_inv] at *
else
  by rw [sin, cos, tan, complex.tan, ← of_real_inj, div_eq_mul_inv, mul_re];
  simp [norm_sq, (div_div_eq_div_mul _ _ _).symm, div_self h]; refl
```

A.4.4 sym2.is_diag_iff_proj_eq

The proof of this lemma is longer than the ground truth and was not contributed to `mathlib`, but we describe it here because the proof is original and includes a nontrivial instantiation of an existential quantifier.

```
theorem sym2.is_diag_iff_proj_eq (z : α × α) :
  is_diag ⟦z⟧ ↔ z.1 = z.2 :=
begin
    intros,
    simp only [is_diag, prod.ext_iff, quot.exists_rep, iff_true, not_true,
    eq_self_iff_true],
    simp [diag], split,
    { rintros ⟨y, hy⟩, cases hy; refl },
    intro h, cases z, existsi z_snd,
    cases h, refl,
end
```

Before `existsi z_snd`, the goal state is

```
z_fst z_snd: α
h: (z_fst, z_snd).fst = (z_fst, z_snd).snd
⊢ ∃ (y : α), (y, y) ≈ (z_fst, z_snd)
```

This goal state never appeared in `mathlib`.

A.4.5 norm_le_zero_iff

The following proof is remarkable because it uses fewer tactic steps and takes a different route to the proof than the ground truth, uses a complex idiom `simpa [...] using @...`, and was predicted in one shot.

```
lemma norm_le_zero_iff {α : Type u_1} [_inst_1 : normed_group α]
  {g : α} : ||g|| ≤ 0 ↔ g = 0 :=
by { simpa [le_antisymm_iff, norm_nonneg] using @norm_eq_zero α _ g }
-- ground truth:
-- by { rw[←dist_zero_righbt!],
--      exact dist_le_zero }
```

The lemmas supplied between the square brackets are used to simplify the main goal. The lemma supplied after the keyword `using` can further simplify the lemmas supplied between the square brackets. The `@` modifier makes all arguments explicit. The string `@norm_eq_zero`

never appeared in our training data but the prediction includes the correct number of correctly typed arguments, and even replaces the second argument with a placeholder `_`, correctly guessing that it can be inferred by the elaborator. Finally, this again showcases the strength of our models as premise selectors: all three lemmas `le_antisymm_iff`, `norm_nonneg`, and `norm_eq_zero` were not used in the human-supplied proof but are necessary for this proof.

Moving forward, we hope that our neural theorem provers will continue to find ways to improve `mathlib` and assist in creating new proofs. More generally, we hope neural theorem proving will one day be become a routine part of the formalization workflow.

A.5 Additional materials for chapter 4

A.5.1 Lean-gym

Below is an example in-terminal trace demonstrating the use of `lean-gym`'s REPL interface:

```
$ lean --run src/repl.lean
["init_search", ["int.prime.dvd_mul", ""]]
{
  "error":null,
  "search_id":"0",
  "tactic_state":"⊢ ∀ {m n : ℤ} {p : ℕ}, nat.prime p →
                   ↑p ∣ m * n → p ∣ m.nat_abs ∨ p ∣ n.nat_abs",
  "tactic_state_id":"0"
}
...
["run_tac",["1","1","apply (nat.prime.dvd_mul hp).mp"]]
{
  "error":null,
  "search_id":"1",
```

```
  "tactic_state":"m n : ℤ, p : ℕ, hp : nat.prime p, h : ↑p ∣ m * n
                  ⊢ p ∣ m.nat_abs * n.nat_abs",
  "tactic_state_id":"2"
}
...
```

A.5.2 Synthetic inequalities

A.5.2.1 List of inequality composition theorems

Below is the list of theorem names from *mathlib* that we use to compose inequalities together. These names are references to theorems in mathlib[1]. One third of the time, we only transform the current composed inequality with one of the following theorems:

- `neg_le_neg`
- `inv_le_inv`
- `mul_self_le_mul_self`
- `div_le_one_of_le`

We otherwise compose the current composed inequality with a newly generated inequality using the following theorems:

- `mul_le_mul`
- `add_le_add`
- `div_le_div`
- `mul_le_mul_of_nonneg`
- `le_mul_of_ratio`

[1] `https://www.github.com/leanprover-community/mathlib/`

A.5.2.2 Examples of synthetic inequalities

We give examples of synthetic inequalities as mentioned in subsubsection 4.0.5.1 generated by composing the above theorems.

A.5.3 $N_D = 0$ $N_S = 0$

Compositions	`AmGm a b (67:ℝ) ((1:ℝ)/(10:ℝ)) ((1:ℝ)/(10:ℝ)) ((8:ℝ)/(10:ℝ))`
Statement	`theorem synthetic_ineq_nb_seed_var_0_depth_0_p_1 (a b : ℝ) (h0 : 0 < a) (h1 : 0 < b) : (67:ℝ) ^ ((8:ℝ) / (10:ℝ)) * b ^ (10:ℝ)⁻¹ * a ^ (10:ℝ)⁻¹ ≤ (8:ℝ) / (10:ℝ) * (67:ℝ) + (10:ℝ)⁻¹ * a + b * (10:ℝ)⁻¹ := sorry`

A.5.4 $N_D = 0$ $N_S = 4$

Compositions	`Sqnonneg a ((a) + ((-68:ℝ)))`

Statement	`theorem synthetic_ineq_nb_seed_var_4_depth_0_p_4`
	`(a b : ℝ)`
	`(h0 : 0 < a)`
	`(h1 : 0 < b) :`
	`(2:ℝ) * (a * (a + -(68:ℝ))) ≤`
	`(a + -(68:ℝ)) ^ 2 + a ^ 2 := sorry`

A.5.5 $N_D = 4$ $N_S = 4$

Compositions	AddLeAdd Bernoulli 99 c AddLeAdd SelfDivConst ((a) / (f)) 6 LeMulOfRatio SelfDivConst c 70 DivLeDiv Cauchy ((a) / (f)) d c (log (((59:ℝ) + f))) Young ((a) / (f)) a ((3:ℝ)/(2:ℝ)) ((3:ℝ)/(1:ℝ))
Statement	theorem synthetic_ineq_nb_seed_var_4_depth_4_p_13 (a b c d e f : ℝ) (h0 : 0 < a) (h1 : 0 < b) (h2 : 0 < c) (h3 : 0 < d) (h4 : 0 < e) (h5 : 0 < f) : (1:ℝ) + (99:ℝ) * c + (a / f / (6:ℝ) + a * (a / f) / ((d ^ 2 + a ^ 2 / f ^ 2) * (real.log ((59:ℝ) + f) ^ 2 + c ^ 2))) ≤ ((a / f) ^ ((3:ℝ) / (2:ℝ)) / ((3:ℝ) / (2:ℝ)) + a ^ 3 / (3:ℝ)) / (real.log ((59:ℝ) + f) * d + a / f * c) ^ 2 * (c / (c / (70:ℝ))) + a / f + (c + (1:ℝ)) ^ 99 := sorry

A.5.6 Example proofs from *mathlib-train*

In this section, we present original proofs found by our models from *mathlib-train*, compared with their ground-truth version.

A.5.7 comap_eq_of_inverse

Statement	lemma comap_eq_of_inverse {f : filter α} {g : filter β} {ϕ : α → β} (ψ : β → α) (eq : ψ ∘ ϕ = id) (hϕ : tendsto ϕ f g) (hψ : tendsto ψ g f) : comap ϕ g = f :=
Ground-truth	begin refine ((comap_mono $ map_le_iff_le_comap.1 hψ).trans _).antisymm (map_le_iff_le_comap.1 hϕ), rw [comap_comap, eq, comap_id], exact le_rfl end
Model proof	begin refine le_antisymm _ (filter.map_le_iff_le_comap.1 hϕ), refine λ s hs, _, rw mem_comap, use [ψ ⁻¹' s, hψ hs], rw [← preimage_comp, eq, preimage_id] end

A.5.8 sum_range_sub_sum_range

Statement	lemma sum_range_sub_sum_range {α : Type*} [add_comm_group α] {f : $\mathbb{N}$ → α} {n m : $\mathbb{N}$} (hnm : n ≤ m) : $\sum$ k in range m, f k - $\sum$ k in range n, f k = $\sum$ k in (range m).filter (λ k, n ≤ k), f k :=
Ground-truth	begin rw [← sum_sdiff (@filter_subset _ (λ k, n ≤ k) _ (range m)), sub_eq_iff_eq_add, ← eq_sub_iff_add_eq, add_sub_cancel'], refine finset.sum_congr (finset.ext $ λ a, ⟨λ h, by simp at *; finish, λ h, have ham : a < m := lt_of_lt_of_le (mem_range.1 h) hnm, by simp * at *⟩) (λ _ _, rfl) end
Model proof	begin rw [← sum_Ico_eq_sub _ hnm], congr, apply finset.ext, simp [Ico.mem, *], tauto end

A.5.9 prod_inv_distrib

Statement	lemma prod_inv_distrib : (∏ x in s, (f x)⁻¹) = (∏ x in s, f x)⁻¹ :=
Ground-truth	begin classical, by_cases h : ∃ x ∈ s, f x = 0, { simpa [prod_eq_zero_iff.mpr h, prod_eq_zero_iff] using h }, { push_neg at h, have h' := prod_ne_zero_iff.mpr h, have hf : ∀ x ∈ s, (f x)⁻¹ * f x = 1 := λ x hx, inv_mul_cancel (h x hx), apply mul_right_cancel' h', simp [h, h', ← finset.prod_mul_distrib, prod_congr rfl hf] } end
Model proof	begin classical; induction s using finset.induction_on with a s has ih, { simp, }, simp only [has, prod_insert has, mul_inv_rev'], finish end

A.5.10 Example proofs from *minif2f-{test,valid,curriculum}*

We present in this section proofs found by our models from *minif2f-{test,valid,curriculum}*, demonstrating some of the capabilities emerging from our training procedure.

A.5.10.1 `imo_1961_p1`

Natural language Solve the system of equations:

$$x + y + z = a$$
$$x^2 + y^2 + z^2 = b^2$$
$$xy = z^2$$

where a and b are constants. Give the conditions that a and b must satisfy so that x, y, z (the solutions of the system) are distinct positive numbers. **Note**: the formalized statement in *miniF2F* is a weaker problem as it focuses on the second part of the question, providing the actual conditions, and asking for a proof that the requirement entails them.

Model proof

```
theorem imo_1961_p1
  (x y z a b : ℝ)
  (h₀ : 0 < x ∧ 0 < y ∧ 0 < z)
  (h₁ : x ≠ y)
  (h₂ : y ≠ z)
  (h₃ : z ≠ x)
  (h₄ : x + y + z = a)
  (h₅ : x^2 + y^2 + z^2 = b^2)
  (h₆ : x * y = z^2) :
  0 < a ∧ b^2 < a^2 ∧ a^2 < 3 * b^2 :=
begin
  revert_all,
  intros,
  rw mul_comm,
```

```
  split,
  { nlinarith [sq_nonneg (x - y), sq_nonneg (y - z)], },
  split,
  { nlinarith [sq_nonneg (z - 1)], },
  revert h₃ h₄,
  field_simp [mul_comm a b],
  rw [mul_comm, ← h₅],
  contrapose!,
  rw mul_comm at h₆,
  rw mul_comm,
  intro h,
  nlinarith [sq_nonneg (x - y), sq_nonneg (y - z)]
end
```

Comments The model is able to close this problem by splitting into cases, contraposing for the last case and using `nlinarith`, which adds some heuristics to an implementation of Fourier Motzkin elimination. It must be noted that the arguments for the first two `nlinarith` uses are not necessary, however the `[sq_nonneg (x - y), sq_nonneg (y - z)]` argument provided on the last line is crucial to close the goal and are completely exogenous (present in no form in the tactic state before).

A.5.10.2 `imo_1964_p2`

Natural language Suppose a, b, c are the sides of a triangle. Prove that

$$a^2(b+c-a)+b^2(c+a-b)+c^2(a+b-c) \leq 3abc$$

Model proof

```
theorem imo_1964_p2
  (a b c : ℝ)
  (h₀ : 0 < a ∧ 0 < b ∧ 0 < c)
  (h₁ : c < a + b)
  (h₂ : b < a + c)
  (h₃ : a < b + c) :
  a^2 * (b + c - a) + b^2 * (c + a - b) + c^2 * (a + b - c)
    ≤ 3 * a * b * c :=
begin
   nlinarith [sq_nonneg (b - a), sq_nonneg (c - b),
     sq_nonneg (a - c), sq_nonneg (c - a)]
end
```

Comments The model is able to close an IMO problem in one-line. It correctly provides exogenous arguments to `nlinarith`, which are necessary to close the goal. Note that either one of the last two arguments in the sequence `[sq_nonneg (b - a), sq_nonneg (c - b), sq_nonneg (a - c), sq_nonneg (c - a)]` can be omitted.

A.5.10.3 `aime_1990_p15`

Natural language Find $ax^5 + by^5$ if the real numbers a, b, x, and y satisfy the equations

$$\begin{aligned} ax + by &= 3, \\ ax^2 + by^2 &= 7, \\ ax^3 + by^3 &= 16, \\ ax^4 + by^4 &= 42. \end{aligned}$$

Note: the formalized statement in *miniF2F* provides the answer and asks for a proof of it.

Model proof

```
theorem aime_1990_p15
  (a b x y : ℝ)
  (h₀ : a * x + b * y = 3)
  (h₁ : a * x^2 + b * y^2 = 7)
  (h₂ : a * x^3 + b * y^3 = 16)
  (h₃ : a * x^4 + b * y^4 = 42) :
  a * x^5 + b * y^5 = 20 :=
begin
  revert_all,
  intros a b x y h₃ h₁ h₂ h₄,
  ring_nf at h₁ h₂,
  rw ← sub_eq_zero at h₁,
  nlinarith [sq_nonneg (x - y), sq_nonneg (a + b - 2),
    sq_nonneg (x + y - (2:ℝ)), sq_nonneg (a - b - 2)]
end
```

Comments The model is able to close a challenging AIME problem by providing crucial exogenous arguments `sq_nonneg (x - y)` and `sq_nonneg (x + y - (2 : ℝ))` to `nlinarith`, which are required to close the goal (while the other two can be removed).

A.5.10.4 `mathd_train_algebra_217`

Natural language Let $f(x) = Ax + B$ and $g(x) = Bx + A$, where $A \neq B$.

If $f(g(x)) - g(f(x)) = B - A$, what is $A + B$? **Note**: the formalized statement in our *curriculum* provides the answer and asks for a proof of it.

Model proof

```
theorem mathd_train_algebra_217
  (a b : ℝ)
  (f g : ℝ → ℝ)
  (h₀ : ∀ x, f x = a * x + b)
  (h₁ : ∀ x, f x = b * x + a)
  (h₂ : a ≠ b)
  (h₃ : ∀ x, f (g x) - g (f x) = b - a) :
  a + b = 0 :=
begin
  revert_all,
  intros a b,
  intros f g,
  contrapose!,
  rintro ⟨h₀, ⟨h₁, h₂⟩⟩,
  use (0 : ℝ),
  simp only [sub_eq_iff_eq_add, h₀, mul_zero] at *,
  simp only [*, zero_add],
  norm_num at h₀
end
```

Comments The model is able to close the goal by contraposing, supplying a witness by the use of `use (0 : ℝ)` and finally leveraging the `simp` and `norm_num`. This example demonstrates the model's ability to chain multiple non-trivial steps of reasoning including the generation of witnesses.

A.5.10.5 `amc12b_2020_p6`

Natural language For all integers $n \geq 9$, the value of

$$\frac{(n+2)! - (n+1)!}{n!}$$

is always which of the following?

(A) a multiple of 4 **(B)** a multiple of 10 **(C)** a prime number

(D) a perfect square **(E)** a perfect cube.

Model proof

```
theorem amc12b_2020_p6
  (n : ℕ)
  (h₀ : 9 ≤ n) :
  ∃ x : ℕ, (x:ℝ)^2 = (nat.factorial (n + 2) -
    nat.factorial (n + 1)) / nat.factorial n :=
begin
  revert_all,
  intros n hn,
  use n + 1,
  field_simp [nat.factorial_ne_zero, pow_succ'],
  ring_exp
end
```

Comments The model exhibits here its ability to implicitly interpret factorial expressions by guessing the correct answer with the use of `use n + 1` and straightforwardly proving its claim right after.

A.5.10.6 mathd_algebra_140

Natural language The expression $24x^2 - 19x - 35$ can be written as $(Ax - 5)(2Bx + C)$, where A, B, and C are positive numbers. Find $AB - 3C$. **Note**: the formalized statement in *miniF2F* provides the answer and asks for a proof of it.

Model proof

```
theorem mathd_algebra_140
  (a b c : ℝ)
  (h₀ : 0 < a ∧ 0 < b ∧ 0 < c)
  (h₁ : ∀ x, 24 * x^2 - 19 * x - 35 =
    (((a * x) - 5) * ((2 * (b * x)) + c))) :
  a * b - 3 * c = -9 :=
begin
  revert_all,
  rintro a b c h₁,
  rw mul_comm,
  rw ←sub_eq_zero,
  field_simp,
  rw sub_eq_add_neg,
  rw ← eq_neg_iff_add_eq_zero,
  rw [mul_comm, ← sub_eq_add_neg, sub_eq_add_neg,
  ← sub_eq_add_neg],
  rw ←sub_eq_zero,
  simp only [mul_add, neg_mul_eq_neg_mul_symm, mul_comm,
  add_left_comm],
  norm_num,
  simp only [pow_two],
  intro h₀,
  rw ← sub_eq_zero,
  linarith [h₁.2, h₁.1, h₀ 3, h₀ 1, h₀ 2]
```

```
end
```

Comments The model mostly "struggles" to make progress up to the last line. There, it presents us with the idea to specialize the statement given in $\texttt{h}_1$ (which became $\texttt{h}_0$ in the process of the proof) at three consecutive natural numbers $1, 2, 3$ which closes the goal with `nlinarith`. This proof is interesting as it demonstrates the model's ability to evaluate symbolic expressions implicitly.

A.5.10.7 `aime_1984_p1`

Natural language Find the value of $a_2 + a_4 + a_6 + a_8 + \ldots + a_{98}$ if a_1, a_2, $a_3 \ldots$ is an arithmetic progression with common difference 1, and $a_1 + a_2 + a_3 + \ldots + a_{98} = 137$. **Note**: the formalized statement in *miniF2F* provides the answer and asks for a proof of it.

Model proof

```
theorem aime_1984_p1
  (u : ℕ → ℚ)
  (h₀ : ∀ n, u (n + 1) = u n + 1)
  (h₁ : ∑ k in finset.range 98, u k.succ = 137) :
  ∑ k in finset.range 49, u (2 * k.succ) = 93 :=
begin
  revert_all,
  simp [finset.sum_range_succ],
  ring,
  simp [pow_succ, mul_comm, add_left_comm, add_assoc],
  simp [two_mul],
  rintro p,
  revert p,
  simp [add_comm],
  intros p hp,
  simp [hp], ring_nf,
  intros,
  nlinarith
end
```

Comments The model finds a rather short proof of the statement compared to typical human formalizations (link).

A.5.10.8 aopsbook_v2_c8_ex1

Natural language Consider the sequence $\{\frac{n}{n+1}\} = \frac{1}{2}, \frac{2}{3}, \frac{3}{4}$, etc. Prove that this sequence tends to 1 as $n \to \infty$

Model proof

```
theorem aopsbook_v2_c8_ex1
  (u : ℕ → ℝ)
  (h₀ : ∀ n, u n = n / (n + 1)) :
  filter.tendsto u filter.at_top (𝓝 1) :=
begin
  revert_all,
  simp [← nnreal.coe_one],
  norm_cast,
  intros,
  revert h₀,
  assume h,
  simp [tendsto_const_nhds, ← nnreal.coe_one, h],
  revert u,
  assume f,
  norm_num,
  rw tendsto_iff_norm_tendsto_zero,
  assume H,
  convert tendsto_norm_zero.comp
    tendsto_one_div_add_at_top_nhds_0_nat,
  funext n,
  have h₀ : (((n:ℝ) + 1) : ℝ) ≠ 0,
  { norm_cast, exact n.succ_ne_zero, },
  rw H,
  field_simp [h₀, norm_neg],
  ring,
```

```
  rw [← sub_eq_zero],
  simp [← sub_eq_zero],
  simp [sub_eq_add_neg],
end
```

Comments An interesting example of a generated `have` statement, denoted by $\mathtt{h}_0$, which is introduced as a cut in order to simplify the expression containing divisions by using `field_simp[`$\mathtt{h}_0$`, norm_neg]` at a later step.

A.5.10.9 `mathd_numbertheory_447`

Natural language What is the sum of the units digits of all the multiples of 3 between 0 and 50? **Note**: the formalized statement in *miniF2F* provides the answer and asks for a proof of it.

Model proof

```
theorem mathd_numbertheory_447 :
  ∑ k in finset.filter (λ x, 3∣x)
  (finset.erase (finset.range 50) 0), (k % 10) = 78 :=
begin
  refl
end
```

Comments Because the predicate `λ x, 3∣x` is registered as decidable over $\mathbb{N}$, we can state the problem by using `finset.filter`, which is computable. Hence, `refl` is able to close the goal.

Bibliography

[1] Jesse Michael Han, Jason Rute, Yuhuai Wu, Edward W Ayers, and Stanislas Polu. Proof artifact co-training for theorem proving with language models. *arXiv preprint arXiv:2102.06203*, 2021.

[2] Kunhao Zheng, Jesse Michael Han, and Stanislas Polu. Minif2f: a cross-system benchmark for formal olympiad-level mathematics. *arXiv preprint arXiv:2109.00110*, 2021.

[3] Stanislas Polu and Ilya Sutskever. Generative language modeling for automated theorem proving. *arXiv preprint arXiv:2009.03393*, 2020.

[4] Thomas C. Hales. Formal proof. *Notices of the AMS*, 55(11):1370–1380, 2008.

[5] The Lean theorem prover authors. The Lean theorem prover authors.

[6] Gabriel Ebner, Sebastian Ullrich, Jared Roesch, Jeremy Avigad, and Leonardo de Moura. A metaprogramming framework for formal verification. *Proc. ACM Program. Lang.*, 1(ICFP):34:1–34:29, 2017.

[7] Mario Caneiro. The type theory of Lean.

[8] Gilles Barthe and Thierry Coquand. An introduction to dependent type theory. In Gilles Barthe, Peter Dybjer, Luís Pinto, and João Saraiva, editors, *Applied Semantics, International Summer School, APPSEM 2000, Caminha, Portugal, September 9-15, 2000, Advanced Lectures*, volume 2395 of *Lecture Notes in Computer Science*, pages 1–41. Springer, 2000.

[9] John Harrison. HOL Light: An Overview. In *International Conference on Theorem Proving in Higher Order Logics*, pages 60–66. Springer, 2009.

[10] Thomas C. Hales, Mark Adams, Gertrud Bauer, Dat Tat Dang, John Harrison, Truong Le Hoang, Cezary Kaliszyk, Victor Magron, Sean McLaughlin, Thang Tat Nguyen, Truong Quang Nguyen, Tobias Nipkow, Steven Obua, Joseph Pleso, Jason M. Rute, Alexey Solovyev, An Hoai Thi Ta, Trung Nam Tran, Diep Thi Trieu, Josef Urban, Ky Khac Vu, and Roland Zumkeller. A formal proof of the kepler conjecture. *CoRR*, abs/1501.02155, 2015.

[11] John Harrison. Formal verification in industry. *Intel Corporation*, 1999.

[12] Georges Gonthier, Andrea Asperti, Jeremy Avigad, Yves Bertot, Cyril Cohen, François Garillot, Stéphane Le Roux, Assia Mahboubi, Russell O'Connor, Sidi Ould Biha, et al. A machine-checked proof of the odd order theorem. In *International Conference on Interactive Theorem Proving*, pages 163–179. Springer, 2013.

[13] Gerwin Klein, June Andronick, Kevin Elphinstone, Gernot Heiser, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. sel4: formal verification of an operating-system kernel. *Commun. ACM*, 53(6):107–115, 2010.

[14] Norman D. Megill and David A. Wheeler. *Metamath: A Computer Language for Pure Mathematics*, 2019. `http://us.metamath.org/downloads/metamath.pdf`.

[15] Jeremy Avigad. Understanding, formal verification, and the philosophy of mathematics. *Journal of the Indian Council of Philosophical Research*, 27:161, 2010.

[16] Jeremy Avigad and John Harrison. Formally verified mathematics. *Commun. ACM*, 57(4):66–75, 2014.

[17] Edward Ayers. *A Tool for Producing Verified, Explainable Proofs*. PhD thesis, University of Cambridge, 2022.

[18] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need. In Isabelle Guyon, Ulrike von Luxburg, Samy Bengio, Hanna M. Wallach, Rob Fergus, S. V. N. Vishwanathan, and Roman Garnett, editors, *Advances in Neural Information Processing Systems 30: Annual Conference on Neural Information Processing Systems 2017, December 4-9, 2017, Long Beach, CA, USA*, pages 5998–6008, 2017.

[19] J. Kiefer and Jacob Wolfowitz. Stochastic estimation of the maximum of a regression function. *Annals of Mathematical Statistics*, 23:462–466, 1952.

[20] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization. In Yoshua Bengio and Yann LeCun, editors, *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*, 2015.

[21] Alec Radford, Karthik Narasimhan, Tim Salimans, and Ilya Sutskever. Improving language understanding by generative pre-training. *URL https://s3-us-west-2. amazonaws. com/openai-assets/researchcovers/languageunsupervised/language understanding paper. pdf*, 2018.

[22] Alec Radford, Jeff Wu, Rewon Child, David Luan, Dario Amodei, and Ilya Sutskever. Language models are unsupervised multitask learners. 2019.

[23] Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. Language models are few-shot learners. In Hugo Larochelle, Marc'Aurelio Ranzato,

Raia Hadsell, Maria-Florina Balcan, and Hsuan-Tien Lin, editors, *Advances in Neural Information Processing Systems 33: Annual Conference on Neural Information Processing Systems 2020, NeurIPS 2020, December 6-12, 2020, virtual*, 2020.

[24] Jared Kaplan, Sam McCandlish, Tom Henighan, Tom B Brown, Benjamin Chess, Rewon Child, Scott Gray, Alec Radford, Jeffrey Wu, and Dario Amodei. Scaling laws for neural language models. *arXiv preprint arXiv:2001.08361*, 2020.

[25] Markus N. Rabe, Dennis Lee, Kshitij Bansal, and Christian Szegedy. Language modeling for formal mathematics. *CoRR*, abs/2006.04757, 2020.

[26] Stanislas Polu and Ilya Sutskever. Generative language modeling for automated theorem proving. *CoRR*, abs/2009.03393, 2020.

[27] Eser Aygün, Zafarali Ahmed, Ankit Anand, Vlad Firoiu, Xavier Glorot, Laurent Orseau, Doina Precup, and Shibl Mourad. Learning to prove from synthetic theorems. *CoRR*, abs/2006.11259, 2020.

[28] Vlad Firoiu, Eser Aygün, Ankit Anand, Zafarali Ahmed, Xavier Glorot, Laurent Orseau, Lei Zhang, Doina Precup, and Shibl Mourad. Training a first-order theorem prover from synthetic data. *CoRR*, abs/2103.03798, 2021.

[29] Yuhuai Wu, Albert Jiang, Jimmy Ba, and Roger B. Grosse. INT: an inequality benchmark for evaluating generalization in theorem proving. *CoRR*, abs/2007.02924, 2020.

[30] Yuhuai Wu, Markus N. Rabe, Wenda Li, Jimmy Ba, Roger B. Grosse, and Christian Szegedy. LIME: learning inductive bias for primitives of mathematical reasoning. *CoRR*, abs/2101.06223, 2021.

[31] Karel Chvalovskỳ, Thibault Gauthier, and Josef Urban. First experiments with data driven conjecturing. 2019.

[32] Josef Urban and Jan Jakubuv. First neural conjecturing datasets and experiments. *CoRR*, abs/2005.14664, 2020.

[33] Cezary Kaliszyk, Josef Urban, Henryk Michalewski, and Mirek Olsák. Reinforcement learning of theorem proving. *CoRR*, abs/1805.07563, 2018.

[34] Jan Jakubuv and Josef Urban. Hammering Mizar by learning clause guidance. *CoRR*, abs/1904.01677, 2019.

[35] Bartosz Piotrowski and Josef Urban. Atpboost: Learning premise selection in binary setting with ATP feedback. *CoRR*, abs/1802.03375, 2018.

[36] Mingzhe Wang and Jia Deng. Learning to prove theorems by learning to generate theorems. *CoRR*, abs/2002.07019, 2020.

[37] Thibault Gauthier, Cezary Kaliszyk, Josef Urban, Ramana Kumar, and Michael Norrish. Tactictoe: Learning to prove with tactics. *J. Autom. Reason.*, 65(2):257–286, 2021.

[38] Lasse Blaauwbroek, Josef Urban, and Herman Geuvers. The tactician - A seamless, interactive tactic learner and prover for coq. In Christoph Benzmüller and Bruce R. Miller, editors, *Intelligent Computer Mathematics - 13th International Conference, CICM 2020, Bertinoro, Italy, July 26-31, 2020, Proceedings*, volume 12236 of *Lecture Notes in Computer Science*, pages 271–277. Springer, 2020.

[39] Kshitij Bansal, Sarah M. Loos, Markus N. Rabe, Christian Szegedy, and Stewart Wilcox. Holist: An environment for machine learning of higher order logic theorem proving. In Kamalika Chaudhuri and Ruslan Salakhutdinov, editors, *Proceedings of the 36th International Conference on Machine Learning, ICML 2019, 9-15 June 2019, Long Beach, California, USA*, volume 97 of *Proceedings of Machine Learning Research*, pages 454–463. PMLR, 2019.

[40] Kshitij Bansal, Sarah M. Loos, Markus N. Rabe, and Christian Szegedy. Learning to reason in large theories without imitation. *CoRR*, abs/1905.10501, 2019.

[41] Aditya Paliwal, Sarah M. Loos, Markus N. Rabe, Kshitij Bansal, and Christian Szegedy. Graph representations for higher-order logic and theorem proving. In *The Thirty-Fourth AAAI Conference on Artificial Intelligence, AAAI 2020, The Thirty-Second Innovative Applications of Artificial Intelligence Conference, IAAI 2020, The Tenth AAAI Symposium on Educational Advances in Artificial Intelligence, EAAI 2020, New York, NY, USA, February 7-12, 2020*, pages 2967–2974. AAAI Press, 2020.

[42] Minchao Wu, Michael Norrish, Christian Walder, and Amir Dezfouli. Tacticzero: Learning to prove theorems from scratch with deep reinforcement learning. In A. Beygelzimer, Y. Dauphin, P. Liang, and J. Wortman Vaughan, editors, *Advances in Neural Information Processing Systems*, 2021.

[43] Kaiyu Yang and Jia Deng. Learning to prove theorems via interacting with proof assistants. In Kamalika Chaudhuri and Ruslan Salakhutdinov, editors, *Proceedings of the 36th International Conference on Machine Learning, ICML 2019, 9-15 June 2019, Long Beach, California, USA*, volume 97 of *Proceedings of Machine Learning Research*, pages 6984–6994. PMLR, 2019.

[44] Alex Sanchez-Stern, Yousef Alhessi, Lawrence K. Saul, and Sorin Lerner. Generating correctness proofs with neural networks. In Koushik Sen and Mayur Naik, editors, *Proceedings of the 4th ACM SIGPLAN International Workshop on Machine Learning and Programming Languages, MAPL@PLDI 2020, London, UK, June 15, 2020*, pages 1–10. ACM, 2020.

[45] Josef Urban and Jan Jakubuv. First neural conjecturing datasets and experiments. In Christoph Benzmüller and Bruce R. Miller, editors, *Intelligent Computer Mathematics - 13th International Conference, CICM 2020, Bertinoro, Italy, July 26-31, 2020,*

Proceedings, volume 12236 of *Lecture Notes in Computer Science*, pages 315–323. Springer, 2020.

[46] Daniel Whalen. Holophrasm: a neural automated theorem prover for higher-order logic. *CoRR*, abs/1608.02644, 2016.

[47] Mingzhe Wang and Jia Deng. Learning to prove theorems by learning to generate theorems. In Hugo Larochelle, Marc'Aurelio Ranzato, Raia Hadsell, Maria-Florina Balcan, and Hsuan-Tien Lin, editors, *Advances in Neural Information Processing Systems 33: Annual Conference on Neural Information Processing Systems 2020, NeurIPS 2020, December 6-12, 2020, virtual*, 2020.

[48] Stanislas Polu and Ilya Sutskever. Generative language modeling for automated theorem proving. *CoRR*, abs/2009.03393, 2020.

[49] Jesse Michael Han, Jason Rute, Yuhuai Wu, Edward W. Ayers, and Stanislas Polu. Proof artifact co-training for theorem proving with language models. *CoRR*, abs/2102.06203, 2021.

[50] Freek Wiedijk. The De Bruijn factor, 2000.

[51] Alexey Dosovitskiy, Lucas Beyer, Alexander Kolesnikov, Dirk Weissenborn, Xiaohua Zhai, Thomas Unterthiner, Mostafa Dehghani, Matthias Minderer, Georg Heigold, Sylvain Gelly, Jakob Uszkoreit, and Neil Houlsby. An image is worth 16x16 words: Transformers for image recognition at scale. In *9th International Conference on Learning Representations, ICLR 2021, Virtual Event, Austria, May 3-7, 2021*. OpenReview.net, 2021.

[52] Alec Radford, Jong Wook Kim, Chris Hallacy, Aditya Ramesh, Gabriel Goh, Sandhini Agarwal, Girish Sastry, Amanda Askell, Pamela Mishkin, Jack Clark, Gretchen Krueger, and Ilya Sutskever. Learning transferable visual models from natural language supervision. In Marina Meila and Tong Zhang, editors, *Proceedings of the 38th International Conference on Machine Learning, ICML 2021, 18-24 July 2021, Virtual Event*, volume 139 of *Proceedings of Machine Learning Research*, pages 8748–8763. PMLR, 2021.

[53] Tom Henighan, Jared Kaplan, Mor Katz, Mark Chen, Christopher Hesse, Jacob Jackson, Heewoo Jun, Tom B. Brown, Prafulla Dhariwal, Scott Gray, Chris Hallacy, Benjamin Mann, Alec Radford, Aditya Ramesh, Nick Ryder, Daniel M. Ziegler, John Schulman, Dario Amodei, and Sam McCandlish. Scaling laws for autoregressive generative modeling. *CoRR*, abs/2010.14701, 2020.

[54] Jared Kaplan, Sam McCandlish, Tom Henighan, Tom B. Brown, Benjamin Chess, Rewon Child, Scott Gray, Alec Radford, Jeffrey Wu, and Dario Amodei. Scaling laws for neural language models. *CoRR*, abs/2001.08361, 2020.

[55] Leonardo Mendonça de Moura, Soonho Kong, Jeremy Avigad, Floris van Doorn, and Jakob von Raumer. The Lean theorem prover (system description). In Amy P. Felty and Aart Middeldorp, editors, *Automated Deduction - CADE-25 - 25th International Conference on Automated Deduction, Berlin, Germany, August 1-7, 2015, Proceedings*, volume 9195 of *Lecture Notes in Computer Science*, pages 378–388. Springer, 2015.

[56] Kevin Buzzard, Chris Hughes, Kenny Lau, Amelia Livingston, Ramon Fernández Mir, and Scott Morrison. Schemes in Lean. *Experimental Mathematics*, 0(0):1–9, 2021.

[57] Jesse Michael Han and Floris van Doorn. A formal proof of the independence of the continuum hypothesis. In Jasmin Blanchette and Catalin Hritcu, editors, *Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs, CPP 2020, New Orleans, LA, USA, January 20-21, 2020*, pages 353–366. ACM, 2020.

[58] Kevin Buzzard, Johan Commelin, and Patrick Massot. Formalising perfectoid spaces. In Jasmin Blanchette and Catalin Hritcu, editors, *Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs, CPP 2020, New Orleans, LA, USA, January 20-21, 2020*, pages 299–312. ACM, 2020.

[59] Peter Scholze. Liquid tensor experiment. `https://xenaproject.wordpress.com/2020/12/05/liquid-tensor-experiment/`, 2020. Formalization available at `https://github.com/leanprover-community/lean-liquid`.

[60] Frank Pfenning and Christine Paulin-Mohring. Inductively defined types in the calculus of constructions. In Michael G. Main, Austin Melton, Michael W. Mislove, and David A. Schmidt, editors, *Mathematical Foundations of Programming Semantics, 5th International Conference, Tulane University, New Orleans, Louisiana, USA, March 29 - April 1, 1989, Proceedings*, volume 442 of *Lecture Notes in Computer Science*, pages 209–228. Springer, 1989.

[61] mathlib. The lean mathematical library. In Jasmin Blanchette and Catalin Hritcu, editors, *Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs, CPP 2020, New Orleans, LA, USA, January 20-21, 2020*, pages 367–381. ACM, 2020.

[62] Guillaume Lample and François Charton. Deep learning for symbolic mathematics. In *8th International Conference on Learning Representations, ICLR 2020, Addis Ababa, Ethiopia, April 26-30, 2020*. OpenReview.net, 2020.

[63] François Charton, Amaury Hayat, and Guillaume Lample. Learning advanced mathematical computations from examples. In *9th International Conference on Learning Representations, ICLR 2021, Virtual Event, Austria, May 3-7, 2021*. OpenReview.net, 2021.

[64] Christopher Hahn, Frederik Schmitt, Jens U. Kreber, Markus Norman Rabe, and Bernd Finkbeiner. Teaching temporal logics to neural networks. In *9th International*

Conference on Learning Representations, ICLR 2021, Virtual Event, Austria, May 3-7, 2021. OpenReview.net, 2021.

[65] Wenda Li, Lei Yu, Yuhuai Wu, and Lawrence C. Paulson. Isarstep: a benchmark for high-level mathematical reasoning. In *International Conference on Learning Representations*, 2021.

[66] Markus Norman Rabe, Dennis Lee, Kshitij Bansal, and Christian Szegedy. Mathematical reasoning via self-supervised skip-tree training. In *9th International Conference on Learning Representations, ICLR 2021, Virtual Event, Austria, May 3-7, 2021*. OpenReview.net, 2021.

[67] Yuhuai Wu, Markus N. Rabe, Wenda Li, Jimmy Ba, Roger B. Grosse, and Christian Szegedy. LIME: learning inductive bias for primitives of mathematical reasoning. In Marina Meila and Tong Zhang, editors, *Proceedings of the 38th International Conference on Machine Learning, ICML 2021, 18-24 July 2021, Virtual Event*, volume 139 of *Proceedings of Machine Learning Research*, pages 11251–11262. PMLR, 2021.

[68] Cezary Kaliszyk and Josef Urban. Learning-assisted theorem proving with millions of lemmas. *J. Symb. Comput.*, 69:109–128, 2015.

[69] Cezary Kaliszyk, Josef Urban, and Jirí Vyskocil. Lemmatization for stronger reasoning in large theories. In Carsten Lutz and Silvio Ranise, editors, *Frontiers of Combining Systems - 10th International Symposium, FroCoS 2015, Wroclaw, Poland, September 21-24, 2015. Proceedings*, volume 9322 of *Lecture Notes in Computer Science*, pages 341–356. Springer, 2015.

[70] Pengyu Nie, Karl Palmskog, Junyi Jessy Li, and Milos Gligoric. Deep generation of coq lemma names using elaborated terms. In Nicolas Peltier and Viorica Sofronie-Stokkermans, editors, *Automated Reasoning - 10th International Joint Conference, IJCAR 2020, Paris, France, July 1-4, 2020, Proceedings, Part II*, volume 12167 of *Lecture Notes in Computer Science*, pages 97–118. Springer, 2020.

[71] Cezary Kaliszyk, François Chollet, and Christian Szegedy. Holstep: A machine learning dataset for higher-order logic theorem proving. In *5th International Conference on Learning Representations, ICLR 2017, Toulon, France, April 24-26, 2017, Conference Track Proceedings*. OpenReview.net, 2017.

[72] Daniel Huang, Prafulla Dhariwal, Dawn Song, and Ilya Sutskever. Gamepad: A learning environment for theorem proving. In *7th International Conference on Learning Representations, ICLR 2019, New Orleans, LA, USA, May 6-9, 2019*. OpenReview.net, 2019.

[73] Cezary Kaliszyk and Josef Urban. Mizar 40 for Mizar 40. *J. Autom. Reason.*, 55(3):245–256, 2015.

[74] Kshitij Bansal, Sarah M. Loos, Markus N. Rabe, Christian Szegedy, and Stewart Wilcox. Holist: An environment for machine learning of higher-order theorem proving (extended version). *CoRR*, abs/1904.03241, 2019.

[75] Joseph Palermo, Edward Ye, and Jesse Michael Han. Synthetic proof term data augmentation for theorem proving with language models. In *Proceedings of the 7th Conference on Artificial Intelligence and Theorem Proving*, 2022. To appear.

[76] William A Howard. The formulae-as-types notion of construction. *To HB Curry: essays on combinatory logic, lambda calculus and formalism*, 44:479–490, 1980.

[77] Gabriel Ebner, Sebastian Ullrich, Jared Roesch, Jeremy Avigad, and Leonardo de Moura. A metaprogramming framework for formal verification. *Proc. ACM Program. Lang.*, 1(ICFP), August 2017.

[78] The mathlib Community. The Lean mathematical library. In *Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs*, CPP 2020, page 367–381, New York, NY, USA, 2020. Association for Computing Machinery.

[79] Steven Y. Feng, Varun Gangal, Jason Wei, Sarath Chandar, Soroush Vosoughi, Teruko Mitamura, and Eduard H. Hovy. A survey of data augmentation approaches for NLP. *CoRR*, abs/2105.03075, 2021.

[80] Yiben Yang, Chaitanya Malaviya, Jared Fernandez, Swabha Swayamdipta, Ronan Le Bras, Ji-Ping Wang, Chandra Bhagavatula, Yejin Choi, and Doug Downey. Generative data augmentation for commonsense reasoning. In *Findings of the Association for Computational Linguistics: EMNLP 2020*, pages 1008–1025, Online, November 2020. Association for Computational Linguistics.

[81] Leonardo de Moura, Soonho Kong, Jeremy Avigad, Floris van Doorn, and Jakob von Raumer. The Lean Theorem Prover (System Description). In Amy P Felty and Aart Middeldorp, editors, *Automated Deduction - CADE-25*, pages 378–388, Cham, 2015. Springer International Publishing.

[82] Juan Ramos et al. Using tf-idf to determine word relevance in document queries. In *Proceedings of the first instructional conference on machine learning*, volume 242, pages 29–48. Citeseer, 2003.

[83] Myle Ott, Sergey Edunov, Alexei Baevski, Angela Fan, Sam Gross, Nathan Ng, David Grangier, and Michael Auli. fairseq: A fast, extensible toolkit for sequence modeling. *CoRR*, abs/1904.01038, 2019.

[84] Ilya Sutskever, Oriol Vinyals, and Quoc V. Le. Sequence to sequence learning with neural networks. In Zoubin Ghahramani, Max Welling, Corinna Cortes, Neil D. Lawrence, and Kilian Q. Weinberger, editors, *Advances in Neural Information Processing Systems 27: Annual Conference on Neural Information Processing Systems 2014, December 8-13 2014, Montreal, Quebec, Canada*, pages 3104–3112, 2014.

[85] Stanislas Polu, Jesse Michael Han, Kunhao Zheng, Mantas Baksys, Igor Babuschkin, and Ilya Sutskever. Formal mathematics statement curriculum learning. *arXiv preprint arXiv:2202.01344*, 2022.

[86] Tom B Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. Language models are few-shot learners. *arXiv preprint arXiv:2005.14165*, 2020.

[87] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. BERT: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805*, 2018.

[88] Yonghui Wu, Mike Schuster, Zhifeng Chen, Quoc V Le, Mohammad Norouzi, Wolfgang Macherey, Maxim Krikun, Yuan Cao, Qin Gao, Klaus Macherey, et al. Google's neural machine translation system: Bridging the gap between human and machine translation. *arXiv preprint arXiv:1609.08144*, 2016.

[89] Alec Radford, Jong Wook Kim, Chris Hallacy, Aditya Ramesh, Gabriel Goh, Sandhini Agarwal, Girish Sastry, Amanda Askell, Pamela Mishkin, Jack Clark, et al. Learning transferable visual models from natural language supervision. *arXiv preprint arXiv:2103.00020*, 2021.

[90] Mingxing Tan and Quoc Le. Efficientnet: Rethinking model scaling for convolutional neural networks. In *International Conference on Machine Learning*, pages 6105–6114. PMLR, 2019.

[91] Aditya Ramesh, Mikhail Pavlov, Gabriel Goh, Scott Gray, Chelsea Voss, Alec Radford, Mark Chen, and Ilya Sutskever. Zero-shot text-to-image generation. *arXiv preprint arXiv:2102.12092*, 2021.

[92] Tero Karras, Samuli Laine, and Timo Aila. A style-based generator architecture for generative adversarial networks. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 4401–4410, 2019.

[93] David Silver, Aja Huang, Chris J Maddison, Arthur Guez, Laurent Sifre, George Van Den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, et al. Mastering the game of go with deep neural networks and tree search. *nature*, 529(7587):484–489, 2016.

[94] David Silver, Thomas Hubert, Julian Schrittwieser, Ioannis Antonoglou, Matthew Lai, Arthur Guez, Marc Lanctot, Laurent Sifre, Dharshan Kumaran, Thore Graepel, et al. Mastering chess and shogi by self-play with a general reinforcement learning algorithm. *arXiv preprint arXiv:1712.01815*, 2017.

[95] Christopher Berner, Greg Brockman, Brooke Chan, Vicki Cheung, Przemysław Dębiak, Christy Dennison, David Farhi, Quirin Fischer, Shariq Hashme, Chris Hesse, et al.

Dota 2 with large scale deep reinforcement learning. *arXiv preprint arXiv:1912.06680*, 2019.

[96] Oriol Vinyals, Igor Babuschkin, Wojciech M Czarnecki, Michaël Mathieu, Andrew Dudzik, Junyoung Chung, David H Choi, Richard Powell, Timo Ewalds, Petko Georgiev, et al. Grandmaster level in starcraft ii using multi-agent reinforcement learning. *Nature*, 575(7782):350–354, 2019.

[97] Cameron B Browne, Edward Powley, Daniel Whitehouse, Simon M Lucas, Peter I Cowling, Philipp Rohlfshagen, Stephen Tavener, Diego Perez, Spyridon Samothrakis, and Simon Colton. A survey of monte carlo tree search methods. *IEEE Transactions on Computational Intelligence and AI in games*, 4(1):1–43, 2012.

[98] Geoffrey Irving, Christian Szegedy, Alexander A Alemi, Niklas Eén, François Chollet, and Josef Urban. DeepMath—deep sequence models for premise selection. In *Advances in Neural Information Processing Systems*, pages 2235–2243, 2016.

[99] Mingzhe Wang, Yihe Tang, Jian Wang, and Jia Deng. Premise selection for theorem proving by deep graph embedding. In *Advances in Neural Information Processing Systems*, pages 2786–2796, 2017.

[100] Daniel Selsam, Matthew Lamm, Benedikt Bünz, Percy Liang, Leonardo de Moura, and David L Dill. Learning a sat solver from single-bit supervision. *arXiv preprint arXiv:1802.03685*, 2018.

[101] Sarah Loos, Geoffrey Irving, Christian Szegedy, and Cezary Kaliszyk. Deep network guided proof search. *arXiv preprint arXiv:1701.06972*, 2017.

[102] Vlad Firoiu, Eser Aygun, Ankit Anand, Zafarali Ahmed, Xavier Glorot, Laurent Orseau, Lei Zhang, Doina Precup, and Shibl Mourad. Training a first-order theorem prover from synthetic data. *arXiv preprint arXiv:2103.03798*, 2021.

[103] Markus N Rabe, Dennis Lee, Kshitij Bansal, and Christian Szegedy. Mathematical reasoning via self-supervised skip-tree training. *arXiv preprint arXiv:2006.04757*, 2020.

[104] Daniel Huang, Prafulla Dhariwal, Dawn Song, and Ilya Sutskever. Gamepad: A learning environment for theorem proving. *arXiv preprint arXiv:1806.00608*, 2018.

[105] Kaiyu Yang and Jia Deng. Learning to prove theorems via interacting with proof assistants. *arXiv preprint arXiv:1905.09381*, 2019.

[106] Josef Urban and Jan Jakubův. First neural conjecturing datasets and experiments. *arXiv preprint arXiv:2005.14664*, 2020.

[107] Wenda Li, Lei Yu, Yuhuai Wu, and Lawrence C Paulson. Modelling high-level mathematical reasoning in mechanised declarative proofs. *arXiv preprint arXiv:2006.09265*, 2020.

[108] Lean theorem prover. `https://leanprover.github.io/about/`.

[109] Kevin Buzzard, Johan Commelin, and Patrick Massot. Lean perfectoid spaces. `https://leanprover-community.github.io/lean-perfectoid-spaces/`, 2019.

[110] Peter Scholze. Liquid tensor experiment. `https://xenaproject.wordpress.com/2020/12/05/liquid-tensor-experiment/`, 2020.

[111] Yuhuai Wu, Albert Qiaochu Jiang, Jimmy Ba, and Roger Grosse. Int: An inequality benchmark for evaluating generalization in theorem proving. *arXiv preprint arXiv:2007.02924*, 2020.

[112] Sandor Lehoczky and Richard Rusczyk. *The Art of Problem Solving, Volume 1: the Basics*. ISBN:978-0-9773045-6-1.

[113] Sandor Lehoczky and Richard Rusczyk. *The Art of Problem Solving, Volume 2: and Beyond*. ISBN:978-0-9773045-8-5.

[114] Dan Hendrycks, Collin Burns, Saurav Kadavath, Akul Arora, Steven Basart, Eric Tang, Dawn Song, and Jacob Steinhardt. Measuring mathematical problem solving with the math dataset. *arXiv preprint arXiv:2103.03874*, 2021.

[115] Konrad Czechowski, Tomasz Odrzygóźdź, Marek Zbysiński, Michał Zawalski, Krzysztof Olejnik, Yuhuai Wu, Lukasz Kucinski, and Piotr Miłoś. Subgoal search for complex reasoning tasks. *Advances in Neural Information Processing Systems*, 34, 2021.

[116] Thibault Gauthier and Cezary Kaliszyk. Sharing HOL4 and HOL light proof knowledge. In Martin Davis, Ansgar Fehnker, Annabelle McIver, and Andrei Voronkov, editors, *Logic for Programming, Artificial Intelligence, and Reasoning - 20th International Conference, LPAR-20 2015, Suva, Fiji, November 24-28, 2015, Proceedings*, volume 9450 of *Lecture Notes in Computer Science*, pages 372–386. Springer, 2015.

[117] Qingxiang Wang, Cezary Kaliszyk, and Josef Urban. First experiments with neural translation of informal to formal mathematics. In Florian Rabe, William M. Farmer, Grant O. Passmore, and Abdou Youssef, editors, *Intelligent Computer Mathematics - 11th International Conference, CICM 2018, Hagenberg, Austria, August 13-17, 2018, Proceedings*, volume 11006 of *Lecture Notes in Computer Science*, pages 255–270. Springer, 2018.

[118] Qingxiang Wang, Chad E. Brown, Cezary Kaliszyk, and Josef Urban. Exploration of neural machine translation in autoformalization of mathematics in Mizar. In Jasmin Blanchette and Catalin Hritcu, editors, *Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs, CPP 2020, New Orleans, LA, USA, January 20-21, 2020*, pages 85–98. ACM, 2020.

[119] M. Ganesalingam and W. T. Gowers. A fully automatic theorem prover with human-style output. *J. Autom. Reason.*, 58(2):253–291, 2017.