Trinity College Dublin
Coláiste na Tríonóide, Baile Átha Cliath
The University of Dublin

School of Computer Science and Statistics

Applications of Reinforcement Learning to Automated Theorem Proving

Conn Breathnach

Supervisor: Associate Professor Ivana Dusparic

April 2024

A dissertation submitted in partial fulfilment
of the requirements for the degree of
MSc (Computer Science)

Declaration

I hereby declare that this dissertation is entirely my own work and that it has not been submitted as an exercise for a degree at this or any other university.

I have read and I understand the plagiarism provisions in the General Regulations of the University Calendar for the current year, found at `http://www.tcd.ie/calendar`.

I have completed the Online Tutorial on avoiding plagiarism 'Ready Steady Write', located at `http://tcd-ie.libguides.com/plagiarism/ready-steady-write`.

I consent / do not consent to the examiner retaining a copy of the thesis beyond the examining period, should they so wish (EU GDPR May 2018).

Signed:Conn Breathnach Date: 15/04/2024

Abstract

In recent years many fields have had their research accelerated through the addition of computers in the research process. Pure mathematics is one such field, and this combination of computers with traditional mathematical research led to the creation of proof assistants. Proof assistants provide mathematicians with tools to ensure that steps taken in logically proving theorems are consistent and do not contain any errors. The rise in artificial intelligence (AI) methods eventually led to the birth of automated theorem proving (ATP), where computers would solve mathematical theorems without requiring human interaction or guidance.
With the advent of deep learning, computers have become capable at performing highly complex tasks that have previously been seen as impossible for machines to complete. Thanks to this, automated theorem proving is slowly gaining awareness as a potential area for deep learning methods to be applied. Reinforcement learning is a machine learning paradigm that works by learning the optimal action or strategy in certain states or scenarios and has shown to be successful in fields such as game playing and robotics control.
Current state of the art methods for automated theorem proving employ language modelling techniques for automated theorem proving, though these methods are limited by a lack of planning and search. The motivation of this work is to employ reinforcement learning techniques and algorithms as a means of performing automated theorem proving with better strategies.

Acknowledgements

To my supervisor, Prof. Ivana Dusparic for her continuous guidance and supporting me throughout my entire Master's. To my parents and siblings, who have always believed in me and supported my decision to pursue the topics I care about. Finally to my friends, John, Dónal, Joey, Federico and Kristina, who have always been there for me, who helped me in every way throughout this year, and who I can always rely on. I wish them all the best in their own PhD, Master's, and career journeys.

Contents

1 Introduction **1**

1.1 Background . . . 1

1.1.1 Motivation . . . 2

1.2 Research Questions . . . 2

1.3 Thesis Contribution . . . 3

1.4 Thesis Overview . . . 3

2 Related Work **4**

2.1 Automated Theorem Proving . . . 4

2.1.1 Proof Assistants . . . 4

2.1.2 Proof Search . . . 8

2.2 Deep Learning . . . 10

2.2.1 Artificial Neural Networks . . . 10

2.3 Reinforcement Learning . . . 12

2.3.1 Markov Decision Process . . . 12

2.3.2 State-Value Functions . . . 13

2.3.3 Action-Value Functions . . . 13

2.3.4 Optimizing the Bellman Equation . . . 13

2.3.5 Deep Reinforcement Learning . . . 14

2.3.6 Temporal Difference Learning . . . 14

2.3.7 Q-Learning . . . 14

2.3.8 Proximal Policy Optimization . . . 15

2.3.9 Neural Theorem Proving . . . 16

2.4 Language Models . . . 17

2.4.1 Recurrent Neural Networks . . . 17

2.4.2 Transformers and Attention Mechanisms . . . 17

3 Designing Theorem Provers built on Reinforcement Learning **20**

3.1 Overall Architecture . . . 21

3.2 Environment Design . . . 21

3.2.1 States . . . 21
3.2.2 Action Space . . . 22
3.2.3 Reward Function . . . 23
3.3 Large Language Models for Automated Theorem Proving . . . 24
3.3.1 ReProver . . . 24
3.3.2 Encoding text . . . 24
3.4 Interacting with Lean . . . 25
3.4.1 LeanDojo . . . 25
3.5 Reinforcement Learning Models . . . 26
3.5.1 State-Value Model . . . 26
3.5.2 Tactic Generator Policy Gradient Model . . . 26

4 Evaluation **29**
4.1 LeanDojo Benchmark . . . 29
4.1.1 Random Split . . . 29
4.2 Analysis of performance . . . 30
4.3 Limitations and critiques of RL methods . . . 32
4.3.1 Training additional models . . . 32
4.3.2 Reliance on LLMs . . . 33

5 Conclusion **34**
5.1 Thesis Contributions . . . 34
5.2 Future Work . . . 34

A1 Appendix **40**
A1.1 Graphs of training models . . . 40
A1.2 Model Parameters . . . 41
A1.2.1 Parameters for State-Value Model . . . 41
A1.2.2 Parameters for Tactic Generator Model . . . 41
A1.3 Example of LeanDojo datapoint . . . 41

List of Figures

2.1 Goals to be proven at the start of zero_add . . . 6
2.2 Goals to prove using induction . . . 7
2.3 The goal for the inductive step, after proving the base case . . . 7
2.4 Lean asserts that the theorem is proved . . . 7
2.5 Example of a potential proof tree for theorem zero_add. Nodes represent states and edges represent tactics. . . . 9
2.6 Structure of a typical neural network, courtesy of Michael Nielsen . . . 11
2.7 Functions within a neural network, courtesy of Becoming Human . . . 11
2.8 A Markov Decision Process takes an action A_t, and transitions to a new state S_{t+1} with reward R_{t+1} . . . 12
2.9 Structure of a RNN, courtesy of Sushmita Poudel . . . 18
2.10 Transformer architecture, Vaswani et. al . . . 19

3.1 Flow of data between models and Lean . . . 20
3.2 Example of embedded Lean states . . . 22
3.3 Example of reward function for a proved theorem . . . 23
3.4 Example of training a model with that interacts with Lean through LeanDojo 25
3.5 Example of training pass through the state-value model . . . 27
3.6 Architecture of the text generating model . . . 28

4.1 Performance comparison on number of theorems proved . . . 31
4.2 Performance comparison on search (lower is better) . . . 32

A1.1 Loss curve for training of the temporal difference model . . . 40

List of Tables

4.1 Size of LeanDojo dataset splits 29
4.2 Comparison between baseline ReProver and state-value augmented search . . 30
4.3 Comparison between baseline ReProver and state-value augmented search when adjusted for single-step proofs 30

A1.1 State-Value model Parameters 41
A1.2 State-Value model Parameters 41

Nomenclature

RL	Reinforcement Learning
ML	Machine Learning
AI	Artificial Intelligence
DNN	Deep Neural Network
TD	Temporal Difference
V(s)	Value function for state s
Q(s,a)	Q-learning function for action a in state s
R(s, a)	Reward for action a in state s
MDP	Markov Decision Process
π_a	Policy a
γ	Discount factor
$V^*(s, a)$	Optimal Value Function
$Q^*(s, a)$	Optimal Q-Value function
π_*	Optimal Policy

1 Introduction

This thesis explores the application of Reinforcement Learning (RL) to the field of Automated Theorem Proving for improving the capabilities of computers in solving proofs and aiding mathematicians in research-level mathematics. It suggests combining RL with current state-of-the-art methods in order to improve the search capabilities of these models, which is a vital but often overlooked factor in finding proofs for long and complex theorems. It begins by first explaining how reinforcement learning works, and how current methods for automated theorem proving are being implemented. It then explores how to interact with Lean, a programming language with built in proof verification assistance. The implementation of two approaches to combining RL and ATP is then discussed, one for aiding current approaches with a state valuation method trained using RL, and another that approaches the problem from a purely RL-based approach. The final chapters discuss evaluating on these new approaches on standard benchmarks, and the thesis concludes with a critical examination of RL with ATP and potential future work.

1.1 Background

Proving mathematical theorems logically and rigorously is the foundation on which all mathematics is founded, and as theorems become longer and more complex, ensuring that proofs hold no contradiction is of utmost importance. This work was traditionally performed by hand and peer reviewed by other humans, who would attempt to find mistakes or contradictions in a mathematician's proof, and if all logic was found to be consistent and without flaws, the proof would then become a mathematical truth. This process is time-consuming, costly, and prone to error, and this has led to hybrid approaches of humans and computers working together for proof checking, verification, and algorithms to solve mathematical theorems.

The first instance of computers aiding humans in proving a mathematical theorem was the four color theorem [1], where a computer program exhaustively verified the different graph structures relevant to the theorem. While most computer-assisted proofs follow this pattern of proof by exhaustion, recent work has focused on using computers to verify human written

proofs. Automath [2] was the first of its kind, where humans could write proofs in the Automath language and later the computer could verify if the proof was sound.

Modern theorem provers such as Lean [3] have large online communities that work together and build large libraries of theorems which help to improve the usability of the theorem provers themselves. The Lean community has created Mathlib [4], which contains over 140,000 verified theorems. This large-scale gathering and sharing of ideas means that theorems can be added with ease, especially if they need to be built off of previous works. For this reason, Lean is by far one of the most popular proof assistants available.

This thesis employs the use of Lean and Mathlib for its work, as Lean can then be interacted with through the use of the Python programming language, and the quantity and quality of theorems in Mathlib means that it is a rich dataset on which machine learning models can be trained.

1.1.1 Motivation

This aim of this thesis is to explore how automated theorem proving can be improved using reinforcement learning, a machine learning paradigm that has largely been overlooked in research into automated theorem provers. Reinforcement Learning excels in exploring complicated action spaces, as well as planning ahead in order to improve the final outcome of an agent's trajectory. This thesis views automated theorem proving as a reinforcement learning problem, where at each state an agent must choose the correct tactic in order to get closer to a solved proof. When one consider the problem in this way, it is possible to see how an agent that has learned various patterns in state-premise interactions could potentially use these same methods to solve theorems that it has never been exposed to before. Using the LeanDojo [5] benchmark, it is empirically shown that using better methods of exploration and planning based on reinforcement learning can have a positive impact and improve upon current state of the art methods in order to create more powerful theorem provers.

1.2 Research Questions

This thesis explores how Reinforcement Learning approaches can be used to improve the capabilities of automated theorem provers, an approach which has yet to be explored as the field of automated theorem proving is still emerging.

1. Does combining reinforcement learning methods with language models for planning improve performance on benchmarks?
2. How do pure reinforcement learning methods do without pretrained data from language models perform?

3. Can RL methods be used for generating valid text in order to perform theorem proving?

1.3 Thesis Contribution

This thesis identifies a gap in the literature for automated theorem proving where the potential of reinforcement learning for has been overlooked. Specifically, this thesis aims to explore the areas of ATP where reinforcement learning can be used, and where it actually improves upon the current literature. Experiments are performed with various approaches to solve the problem of theorem proving using methods of combining RL with current state-of-the-art methods, particularly in proof search, and analysis is done on the capabilities of RL with and without the support of existing technologies, meaning that in the latter case the RL model performs all actions required in proving a theorem, which includes both state valuation and tactic generation, the latter of which is currently implemented using large language models. The formalization of theorem proving as a Markov Decision Process (MDP) allows for the training of reinforcement learning agents, and the introduction of a state-value model for identifying promising paths in a proof search. Through evaluations between these methods and state-of-the-art methods on a standard baseline, it can be clearly seen that the introduction of RL methods to ATP can have a positive impact on how proof search is performed.

1.4 Thesis Overview

Chapter 2: Background work and literature review

Chapter 3: Experiment setup with LLMs, RL, and Lean

Chapter 4: Training setup and dataset information

Chapter 5: Results

Chapter 6: Conclusion and future work

2 Related Work

This chapter discusses the technical foundations of the research performed in this thesis. It begins by discussing the topic of reinforcement learning, and the algorithms used within this research. It then moves onto the topic of large language models, and how they can be used to train reinforcement learning agents using text as an input. Finally, it discusses how proof assistants work, and how automated theorem provers can be built on these same programs, which are then improved by machine learning methods.

2.1 Automated Theorem Proving

Automated Theorem Proving (ATP) is a subsection of mathematics and computer science which deals with automated reasoning and artificial intelligence in the domain of mathematics. ATP uses computer search to prove mathematical theorems using axioms, proved theorems, and mathematical principles. In recent years, programming languages such as Lean [3], Metamath [6] and Coq [7] have become more prominent as a way for research mathematicians to formalize their work, and perform proof analysis in order to reduce flaws in their handwritten proofs. These languages and the resulting databases of code for proving theorems means that there exists a large swathe of text data corresponding to theorem proving for training machine learning models, and the inherent ability of these programming languages to check, verify, and validify any theorems written in the language ensures that these datasets contain only valid code and high-quality data.

2.1.1 Proof Assistants

Proof assistants refer to programming languages that are used by mathematicians that can perform validation on statements written in the language. By providing a formal and standardized method of writing mathematical statements, it allows for mathematicians to use the power of computers to find logical holes in their arguments. These languages are programming languages, and solving a proof can be seen as equivalent to writing a program in a regular programming language. In writing programs, generative AI tools such as ChatGPT or Copilot have shown some limited capabilities [8], and approaches to Automated

Theorem Proving with generative AI follow similar methods as those used to write programs [9]. The disparity between the two approaches is that proofs written in these languages are more scarce than code for general purpose programming on the internet, and theorems that are applied in proofs are actually built up of their own proofs, but only referenced by name in the new proof.

Lean4

Lean4 is the latest version of the Lean programming language and theorem prover [3], and is one of the most popular languages out there for formalized mathematics. Lean works by having programs interact with the Lean kernel, which verifies if theorems are valid and allows for importing proved theorems from other Lean programs in order to assist in proofs. Lean is based on Calculus of Constructions with inductive types, which is a form of dependant type theory. This allows users to define mathematical objects (such as graphs, natural numbers), and then use these objects in proofs.
The prominence and popularity of Lean as a proof assistant means that a large catalogue of proofs in various mathematical domains has been collected over time. Mathlib4 [4] is a repository containing over 140,000 proofs in Lean4 corresponding to many topics in mathematics such as Abstract Algebra and Group Theory. Mathlib4 is both open source and continuously maintained by the Lean FRO, which means that additions to the repository requires proofs to be valid and follow specific styles for how a proof is written. Thanks to this, machine learning algorithms can learn how to write proofs that are both valid and similar in style to how a human writes a proof. Mathlib4 is therefore a perfect dataset as it contains a vast library of proofs as well as assurances that the proofs within are of high quality and similar to what would be expected of a professional mathematician.

Examples of Lean

Solving a theorem in Lean can be seen as equivalent to writing a program in other software languages, with basic statements and operators in both, which can be used to manipulate user defined types and functions. In Lean, each step of the proof is checked against Lean's kernel, and the proof is either complete ("no goals") or yet to be completed (goals remaining). In a proof, these goals are simply aspects within the theorem that have yet to be proved. In the below example the task is to prove that zero added to a number is equal to the number itself, i.e $0 + n = n$. In the Peano Axioms, $n + 0 = n$ is an axiom, but the ordering matters and $0 + n = n$ is not an axiom, and therefore needs to be proved.
The entire Lean code for this proof is formalized as such:

```
theorem zero_add (n : ℕ) : 0 + n = n := by
    induction n with k k_plus_one
    rw [add_zero]
```

```
    rfl
    rw [add_succ]
    rw [k_plus_one]
    rfl
```

The theorem begins by by defining what needs to be proved.

```
theorem zero_add (n : ℕ) : 0 + n = n := by
```

Comparing this to a regular software program, zero_add would be the function name, and (n : ℕ) would be the function parameters, i.e this theorem makes use of a natural number n. The final part "0 + n = n := by" tells Lean what is needed to be proved, i.e that $0 + n = n$, and would be essentially the expected behaviour of the function.
Now that the theorem has been defined, Lean can now return the goals, which are the unproven aspects of the theorem. Through the course of proving a theorem, the goals will progressively be broken into subgoals, until all theorems can be proven using simple tactics. In this case, Lean tells the user that they are working with a natural number n, and the goal is simply the stated theorem's goal.

Figure 2.1: Goals to be proven at the start of zero_add

In the case of proving zero_add, the induction method can be used to assist in the proof. Recall that proof by induction requires a base case with $n = 0$ to be proved, before then proving an inductive step where a general case k is assumed, and the proof is performed for $k + 1$.
The tactic "induction n with k k_plus_one" can be used to state that induction over n is being performed, with k being the assumption and k_plus_one being the induction step. With the "induction" keyword, Lean splits the goal into two subgoals: one for the case of 0 and another for the inductive case.

Now the bulk of the proof can begin. The "rw" keyword "rewrites" parts of the proof by replacing patterns in the state with different statements. This allows for specific parts of the theorem to be proved, using statements that are already know as fact, or assume to be

Figure 2.2: Goals to prove using induction

(such as the induction hypothesis, which can then later be proved). In this case, it is known that $n + 0 = n$ is an axiom so by calling "rw [add_zero]" on the goal of $0 + 0 = 0$, Lean changes this to $0 = 0$. " rfl " then uses the definition of "relexivity" to finally prove this statement, and it is no longer a goal.

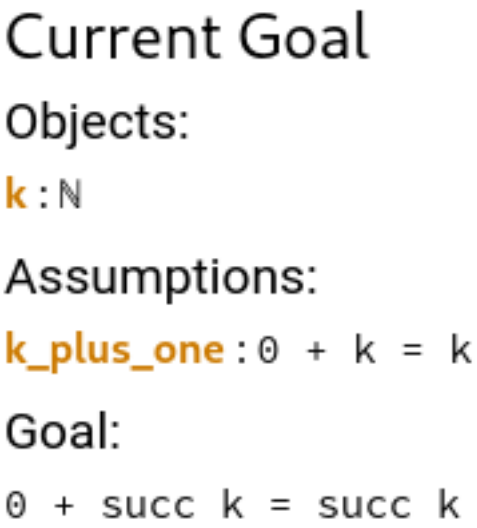


Figure 2.3: The goal for the inductive step, after proving the base case

The rest of the proof is concerned with solving the inductive step, where once again "rw" and " rfl " are applied. After all goals and subgoals have been completed, the proof is complete and Lean confirms this with the statement "No goals"

Figure 2.4: Lean asserts that the theorem is proved

MathLib4 is full of theorems such as the one above, ranging on a broad range of subfields in mathematics. These theorems can become incredibly complex, with proofs becoming many lines long, each line using theorems that themselves are quite long. Being able to notice patterns in a proof, and relate certain states within a proof to other theorems already within Mathlib4 is a problem that requires great understanding of the Mathlib4 library, or at least

the subfield of the theorem being proved. One upside to creating a monolith repository full of open-source code and theorems, as has been done with Mathlib4, is that the contributions to the repository are expected to be of a high quality standard and will follow specific styling guidelines and be written by humans, whose proofs are much more readable than a machine's would be. This allows models to be trained in order to produce Lean4 code that prove theorems that are also written in a manner that is similar to how a human would write. This makes it easier to debug proofs written by AI, and allows a more collaborative approach to ATP between the mathematician and the computer.

2.1.2 Proof Search

The proof to a theorem can be any length, from applying a single tactic to requiring hundreds or thousands of steps, with most step requiring prerequisite theorems to be proved. In order to prove theorems that consist of multiple steps, one must be able to examine potential future states, given the tactics available to use in the current state. In this way, theorem proving can be viewed like a game of chess, albeit without an opponent. Given the current state (current goals to prove, or chess board) and all the actions (tactics, or moves) that can be made, search can be performed through the next states after applying the actions to the given state. Not all tactics will be useful in a certain state, however. Similar to chess, mathematics has too many states to manually explore, so an evaluation function is necessary to reduce the number of searches to only the states that allow a full proof to be reached. Note how in figure 2.5 the tactic `rfl` in this case does not change the state, and a tactic that does not exist will throw an error. It is therefore important that only states that are deemed important are expanded upon.
The problem of theorem proven can now be broken up into two separate parts, both of which are vitally important in solving a proof: tactic generation and state evaluation.

Tactic Generation

Unlike a game such as chess, mathematics has an infinite number of tactics (moves) that can be made in any state. This is trivial to show, as adding any number can be considered a tactic (add 1, add 2, etc are each their own tactics), so simply adding numbers is its own set of tactics.
Many approaches have been used in generating tactics for ATP. Original methods used a hard coded set of tactics and heuristics in order to choose tactics, and these approaches are still powerful and used today, in provers such as Isabelle [10] and Vampire [11]. In recent years with the rise in language models and their abilities in code writing, researchers have been exploring methods of using large language models in the domain of ATP. GPT-f [9] was a generative AI model based on the GPT [12] architecture, and it began a shift away from traditional solvers to a direction involving Neural Networks to suggest tactics. As

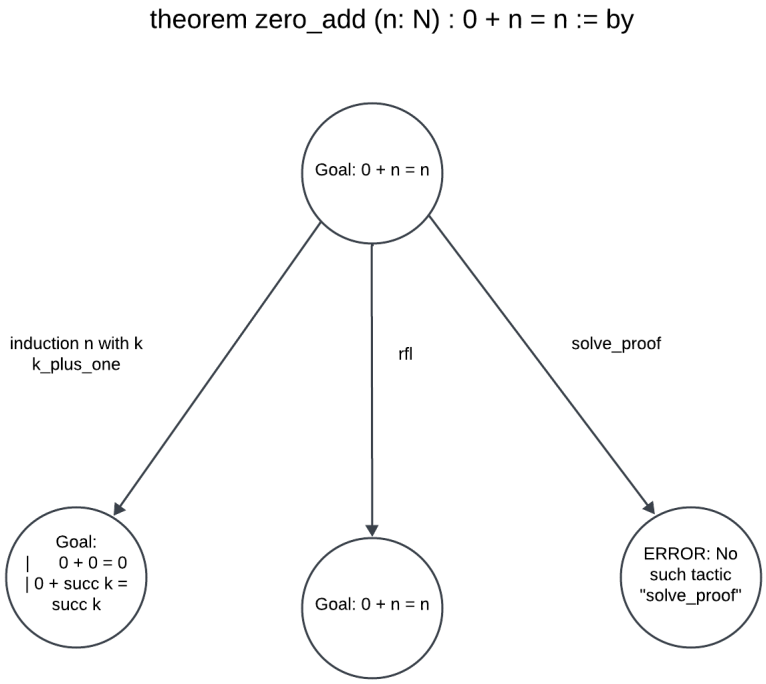


Figure 2.5: Example of a potential proof tree for theorem zero_add. Nodes represent states and edges represent tactics.

interest in the field of ATP grew, more methods were approached with varying levels of success. Approaches such as K-Nearest Neighbour [13] used word embeddings, while Graph2Tac [14] embedded the proof as a dependency tree in order to perform proof search in the Coq theorem prover. While these methods have been successful in expanding the field and performing search using smaller models, they still fail to surpass LLMs on most standardized benchmarks for ATP.

In performing proof search, usually the top-k tactics are applied, and if $k = 1$ the model greedily chooses the perceived best tactic at each step, without viewing any other states aside from the single path taken. Evaluating the best tactic is usually done via a softmax function over all tactics generated, in order to give a probability distribution. This distribution is bootstrapped from the tactic generator itself, so in the case of a LLM, the best tactic is the one with the highest log probability of the tokens generated, given the current state. For RL based methods, it is a similar method, but the distribution of the tactics is taken as the action probabilities sampled from the current policy.

In this regard, RL-based methods for generating for generating tactics without LLMs are evaluated, as well as explore how LLMs can benefit from RL approaches, as discussed in the next section.

State Evaluation

Once tactic candidates for the current proof state have been generated, the top-k are run and the next states for the proof are generated, for each tactic applied. Once these new states are generated, they can be examined in order to find which are the most likely to lead the prover to a full proof. These can then be expanded again, using the tactic generation step, and this process repeats until a proof is found, or the program times out.
For typical LLM applications applied to ATP, a priority queue of tactics is maintained, where the evaluation function is once again based on the log probability of the most likely tactics given the state, sampled from the generator. This particular problem of state evaluation, however, is well defined in the field of RL, and thus came the motivation of applying RL methods to ATP problems, an application that was believed to be well suited for RL. In reinforcement learning, state-value functions are a common framework for solving problems, and allow for models to plan into the future, as the dynamics of an environment can be learned and the value of a state can be estimated by examining expected future states. LLMs struggle with this task of examining later states, as they generate text given the current state, but without planning how the text it generates will impact future states. LLMs, in this way, are only really looking at the immediate next step, and not understanding how the next state relates to solving the overall proof, a methodology that RL excels at.

2.2 Deep Learning

In recent years the field of machine learning has seen remarkable progress and groundbreaking results in a variety of fields thanks to the emergence of deep learning (DL) [15]. Deep learning methods use artificial neural networks (ANNs), also known as deep neural networks (DNNs), and large datasets to find patterns and representations in the data that are typically too complex for linear methods or handcrafted features.

2.2.1 Artificial Neural Networks

Deep Learning methods first emerged in the 1950s with the conception of the perceptron [16], which suggested stacking multiple layers of probabilistic cells together to form multi-layered perceptrons (MLPs), which are the basis for ANNs. These early systems took inspiration from the human brain, where neurons fire based on stimuli, and these neurons can activate other neurons connected to them. By training these networks using large amounts of data, the perceptrons can learn to fire when they encounter patterns in the data that correspond to a positive signal in the training data. In this manner, MLPs can learn representations in the data, even those not explicitly shown in the data and are instead correlated with specific patterns that are unable to be extracted by linear methods or hard-coded as specific features.

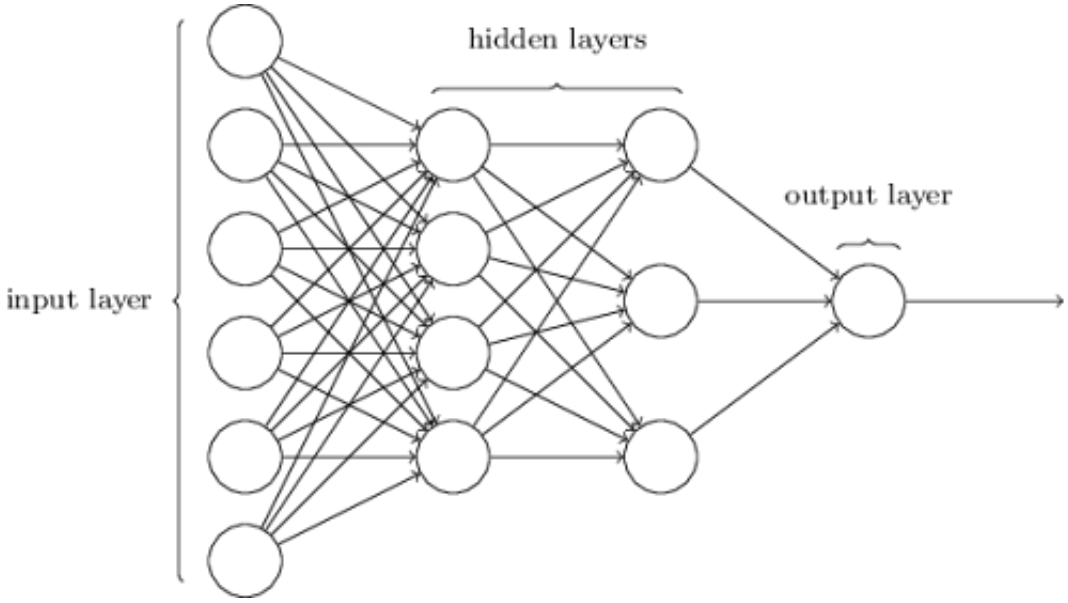
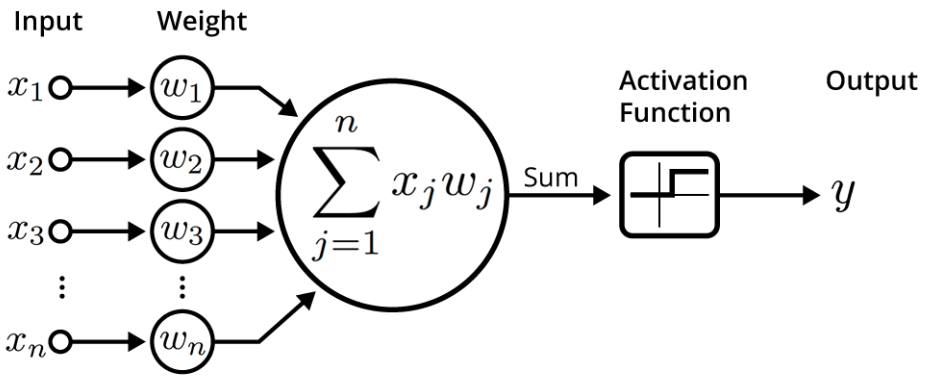


Figure 2.6: Structure of a typical neural network, courtesy of Michael Nielsen



An illustration of an artificial neuron. Source: Becoming Human.

Figure 2.7: Functions within a neural network, courtesy of Becoming Human

Deep neural networks consist of an input layer, which taken in the data to be processed, a number of hidden layers which propagate information through the network, and an output layer, which consists of 1 or more neurons that are used to interpret the output of the network. Each neuron in a layer is connected to its previous and next layer, and each layer computes the weighted sum of its input $\omega x + b$, where ω is the "weight" of the connection between two neurons, x is the sum of the input to the neuron, and b is a bias term. After computing this summation, the neuron passes this value through a nonlinear activation function, which are used to prevent linearity. Without the activation function, the network could be reduced to a simple linear model, and would be unable to encapsulate more complex and nonlinear patterns in the data.

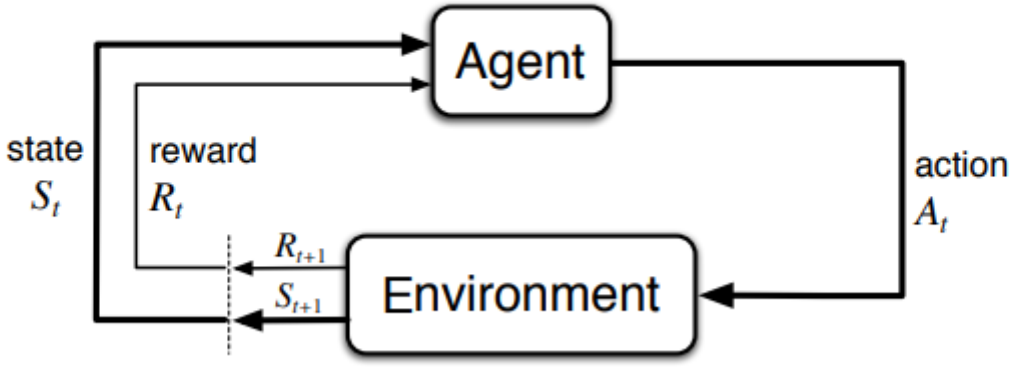


Figure 2.8: A Markov Decision Process takes an action A_t, and transitions to a new state S_{t+1} with reward R_{t+1}

2.3 Reinforcement Learning

Reinforcement Learning [17] is a machine learning approach which has an agent learn actions by having it interact with an environment which provides feedback on how good or bad an action was through a reward signal. Agents learn how to navigate through complex environments through trial-and-error, while also learning about how to interact with the environment in certain states, and how actions cause states to transition, which gives a solid foundation for learning how to act optimally in the long term. The goal of a RL agent is to maximize the cumulative reward throughout the entire episode. A reinforcement learning environment can be formalized as a Markov Decision Process (MDP).

2.3.1 Markov Decision Process

A Markov Decision Process (MDP) is a method of formally structuring a reinforcement learning environment, which consists of a set of states $\mathcal{S}$, actions $\mathcal{A}$, and rewards $\mathcal{R}$, with rewards calculated based on an action $a \in \mathcal{A}$ taken in state $s \in \mathcal{S}$. After this action is taken, the state s transition to a new state $s' \in \mathcal{S}$. The MDP also contains a set of state transition probabilities $\mathcal{P}$ which model how actions allow transition between states with $P_a(s, s') = Pr(s'|s, a)$. A probability of 1 means that the action is deterministic, and a probability of 0 means that in state s, taking action a can never result in the next state being s'. One important aspect of a Markov Decision Process is that all relevant information for a state is within that state itself. This means that for a given state s, one does not need to know anything about the previous states, as all information regarding state s is contained within this state, and is independent of states that appeared before s.

2.3.2 State-Value Functions

In Reinforcement Learning, a Value function refers to a function that estimates the value of either the current state, or an action taken in the current state. These functions are designed in order to optimize the Bellman Equation, which defines the rewards given throughout the environment, with total reward being called the return. The Value function is defined as the value of the current state (given by the reward for the best action), along with a discounted value of the future states that the model is expected to reach. Discounting the future states allows models to better prioritize rewards that are in the immediate future, which are more likely to be earned rather than potential future rewards which are not definitely going to be reached. These functions are parameterized by the policy π.

$$v_\pi(s) = R(s,a) + \gamma \sum_{s'} P(s'|s,a) v_\pi(s') \tag{2.1}$$

The Bellman Equation for the return in state s while following policy π

2.3.3 Action-Value Functions

The State-Action value function (The Q-function) works in a similar manner to the State-Value function, but is tasked with evaluating actions to be taken in various states. This method is also tasked with optimizing the Bellman Equation, and makes use of the State-Value function in order to evaluate future states, similar to the second part of the state-value function itself.

$$q_\pi(s,a) = R(s,a) + \gamma \sum_{s'} P(s'|s,a) v_\pi(s') \tag{2.2}$$

2.3.4 Optimizing the Bellman Equation

The goal of Reinforcement Learning in relation to the Bellman Equation is by maximizing the reward achievable in the environment overall. The notations V^* and Q^* are used to indicate an optimal policy π^* which has optimized the Bellman Equation and can take the best possible action in any state, in order to maximize reward achieved throughout an episode. Optimizing the Bellman Equation can be done through dynamic programming, but often in more complex environments Deep Neural Networks are used to estimate the value functions, as performing dynamic programming across all possible states and actions can be too complex and compute intensive. The Optimal Value Function can be defined as

$$V^*(s) = \max_a \left\{ R(s,a) + \gamma \sum_{s'} P(s'|s,a) V^*(s') \right\} \tag{2.3}$$

The Bellman Equation for the return in state s while following the optimal policy

2.3.5 Deep Reinforcement Learning

Deep Reinforcement Learning is an extension to classical RL where deep neural networks are used to model the parameters of an agent. With Deep Reinforcement Learning the same concepts of maximizing reward in the long term can be used, as well as learning either an optimal policy or a state-action value function, by employing either policy gradient methods or value iteration methods. The difference with Deep Reinforcement Learning is that during updates the parameters of a neural network are modified through backpropagation. Combining deep learning with reinforcement learning allows for models to perform better in complex environments, as neural networks can represent complex states as a simpler combination of features within their hidden states.

2.3.6 Temporal Difference Learning

Temporal Difference (TD) Learning [18] is a reinforcement learning method that uses bootstrapping of the value function in order to estimate future rewards, which allows for better exploration of the environment and improves performance in optimizing the policy. Given a state, TD-Learning follows the current policy and estimates the future rewards given by following this policy, and after reaching the state up to which was estimated, TD-Learning can then compare its estimates to the actual received rewards. The value functions are then optimized in order to improve these estimations by minimizing the loss between the estimates and the actual rewards. The state-value function is updated by

$$v(s) \leftarrow (1 - \alpha v(s)) + \alpha[R(s, a) + \gamma v(s')] \tag{2.4}$$

where α is the learning rate, which corresponds to how drastically out function parameters should be updated, and [R(s, a) + γv(s')] is known as the TD Target, which is the estimate of future rewards. TD-Learning is a powerful method as it can work for any number of future steps, rather than at the end of the episode or only on every step.

2.3.7 Q-Learning

Q-Learning is a reinforcement learning algorithm that learns a state-action value function through interacting with the environment and optimizing the function to maximize the overall reward. Q-Learning makes use of the TD Target in order to optimize between current and future rewards, and by interacting with the environment the Q-Learning algorithm learns to estimate values for a wide range of actions and rewards. Q-Learning excels in small environments with a limited number of states and actions, due to the fact that all states can

be explored, and the memory to store all pairs of states and actions is limited. Issues arise when environments become more complex, as one cannot always explore all states in an environment, and some environments have more states than computationally feasible to store (chess or go, for example).

Algorithm 1 Q-learning Algorithm Overview $Q : \mathcal{X} \times \mathcal{A} \to \mathbb{R}$ courtesy of Martin Thoma

Require:
States $\mathcal{X} = \{1, \dots, n_x\}$
Actions $\mathcal{A} = \{1, \dots, n_a\}$, $A : \mathcal{X} \Rightarrow \mathcal{A}$
Reward function $R : \mathcal{X} \times \mathcal{A} \to \mathbb{R}$
Black-box (probabilistic) transition function $T : \mathcal{X} \times \mathcal{A} \to \mathcal{X}$
Learning rate $\alpha \in [0, 1]$, typically $\alpha = 0.1$
Discounting factor $\gamma \in [0, 1]$
procedure $\mathrm{QLearning}(\mathcal{X}, A, R, T, \alpha, \gamma)$
  Initialize $Q : \mathcal{X} \times \mathcal{A} \to \mathbb{R}$ arbitrarily
  while Q is not converged **do**
    Start in state $s \in \mathcal{X}$
    while s is not terminal **do**
      Calculate π according to Q and exploration strategy (e.g. $\pi(x) \leftarrow_a Q(x, a)$)
      $a \leftarrow \pi(s)$
      $r \leftarrow R(s, a)$ ▷ Receive the reward
      $s' \leftarrow T(s, a)$ ▷ Receive the new state
      $Q(s', a) \leftarrow (1 - \alpha) \cdot Q(s, a) + \alpha \cdot (r + \gamma \cdot \max_{a'} Q(s', a'))$
      $s \leftarrow s'$
    end while
  end while
  return Q
end procedure

Deep Q-Networks

Deep Q-Networks (DQNs) [19] are an implementation of Q-Learning algorithms that use DNNs for computing Q-values for states and actions. This allows models to learn state representations that only approximate states, which is an important factor in complex environments where all states cannot be visited, and therefore it is a powerful tool for learning Q-functions that cannot be learned reasonably without approximation. Deep Q-Networks were a vital factor in achieving 9 Dan performance in the game of Go [20], a game where the state space consists of $\approx 1.7 \times 10^{172}$ states, far more than what is possible to completely explore.

2.3.8 Proximal Policy Optimization

Proximal Policy Optimization [21] is an advanced algorithm which bases its implementation on Policy Gradient methods [22], which are used to train agents to learn in continuous

environments. Given an agent with θ parameters, and π_θ being the policy with θ parameters, gradient ascent can be used in order to maximize reward achieved throughout an episode for an agent. π_θ gives a probability distribution of actions given a state, which makes it more ideal than Q-Learning for continuous action spaces, as Q-Learning estimates are best for discrete action space environments.
PPO is trained in a manner that employs actor-critic methods [23], where it learns both a policy function, known as the actor, and a value function, known as the critic. These two functions are usually approximated via neural networks, and often both functions are learned within the same neural network, so that representations are consistent across both agents. PPO updates using the advantage function $A(a, s)$ rather than the reward estimates, which is calculated by subtracting the episode return from a learned baseline, which is the role of the critic. PPO updates are clipped in order to not drastically change the learned policy, based on earlier works in trust region policy optimization [24].

Algorithm 2 PPO, Actor-Critic Style

1: **for** iteration = 1, 2, ... **do**
2: **for** actor = 1, 2, ..., N **do**
3: Run policy $\pi_{\theta_{\text{old}}}$ in environment for T timesteps
4: Compute advantage estimates $\hat{A}_1, \ldots, \hat{A}_T$
5: **end for**
6: Optimize surrogate L w.r.t. θ, with K epochs and minibatch size $M \leq NT$
7: $\theta_{\text{old}} \leftarrow \theta$
8: **end for**

Here the surrogate L is a loss function that combines the policy loss function, that has been clipped to reduce the update size, and the value function error.

2.3.9 Neural Theorem Proving

Neural Theorem Proving is the combination of neural networks and proof assistants in order to greatly improve the capabilities of traditional proof assistants. GPT-f [9] took inspiration from language generation tasks, and applied them to theorem proving, which allowed for the application of large language models (LLMs), which have seen great success in both language generation and code generation, to theorem proving. LLMs are neural networks on the scale of billions of neurons that are trained on large amounts of unlabelled natural language data, and manage to learn how to generate human like and realistic text.
Since Lean is a programming language as well as a proof assistant, training models that typically generate text on Lean code is a natural step in trying to build theorem provers that can prove more and more theorems. While LLMs cannot perform reasoning steps or

calculations, generating a single step based on the current state is what language models excel at, and so tactic generation can be seen as a language generation task.
While tactic generation is an important part of theorem proving, proof search also requires state evaluation in order to decide which states are more important to expand upon. For this task, language models struggle as it requires a understanding of how the environment interactions work, what future states may look like, and how close a state is to being a completed proof. Current state-of-the-art methods implement beam search [25] as their evaluation metric, which simply chooses the state which has the largest cumulative log probability of already generated text.

2.4 Language Models

Language models are a subset of neural networks which specialize in text data, with tasks such as generation, summarization, and translation. Due to the inherent structure of language, where the ordering of words is of importance and sentences can be variable length, most language models require an architecture that is a modified version of the traditional neural network.

2.4.1 Recurrent Neural Networks

Recurrent Neural Networks (RNNs) [26] are a type of neural network architecture that produces two outputs at each timestep: the regular output that is expected and a hidden state output. RNNs are specially designed to work with timeseries or text data, and this is done by continuously feeding the input and hidden state into the network. The hidden state is used to capture temporal information in the data, which would otherwise be lost without some method of keeping track. This need arises from the fact that slight permutations in a sentence can change the meaning overall. The sentences "Can I walk the dog" and "I can walk the dog" hold two different meanings, and without the network being able to differentiate between the two the output could easily be wrong.
Thanks to the fact that words are fed into the network one at a time, and the network is simply called over and over with an updated hidden state, the RNN model can perceive variable length input, and produce variable length output. This removes the limitation of typical neural networks, who would require a maximum length output which would need to be padded, though it does increase complexity in implementing these models.

2.4.2 Transformers and Attention Mechanisms

In recent years the emergence of the transformer [27] architecture has been behind many of the greatest advances in text generation methods. Transformers make use of attention [28] mechanisms, which learn relations between words without requiring the entire sentence to be

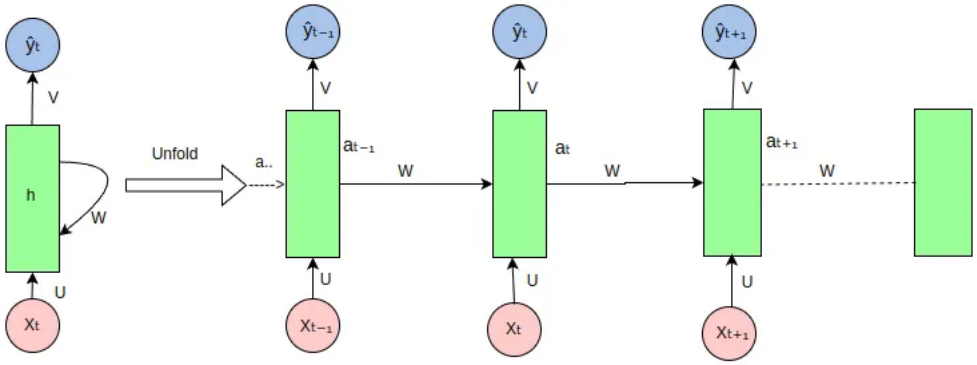


Figure 2.9: Structure of a RNN, courtesy of Sushmita Poudel

fed one word at a time. This fact allows transformers to be trained in parallel at a massive scale, which has led to most of the largest machine learning models being based on the transformer architecture. Most state of the art text generation models in a variety of fields are therefore some form of a transformer, and this includes current state of the art in theorem proving.

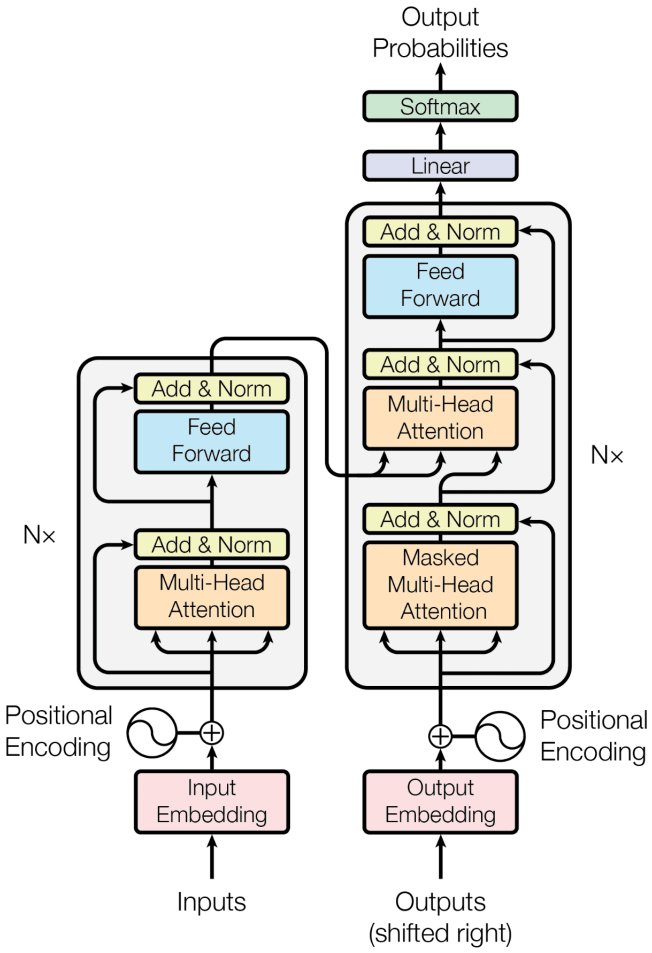


Figure 2.10: Transformer architecture, Vaswani et. al

3 Designing Theorem Provers built on Reinforcement Learning

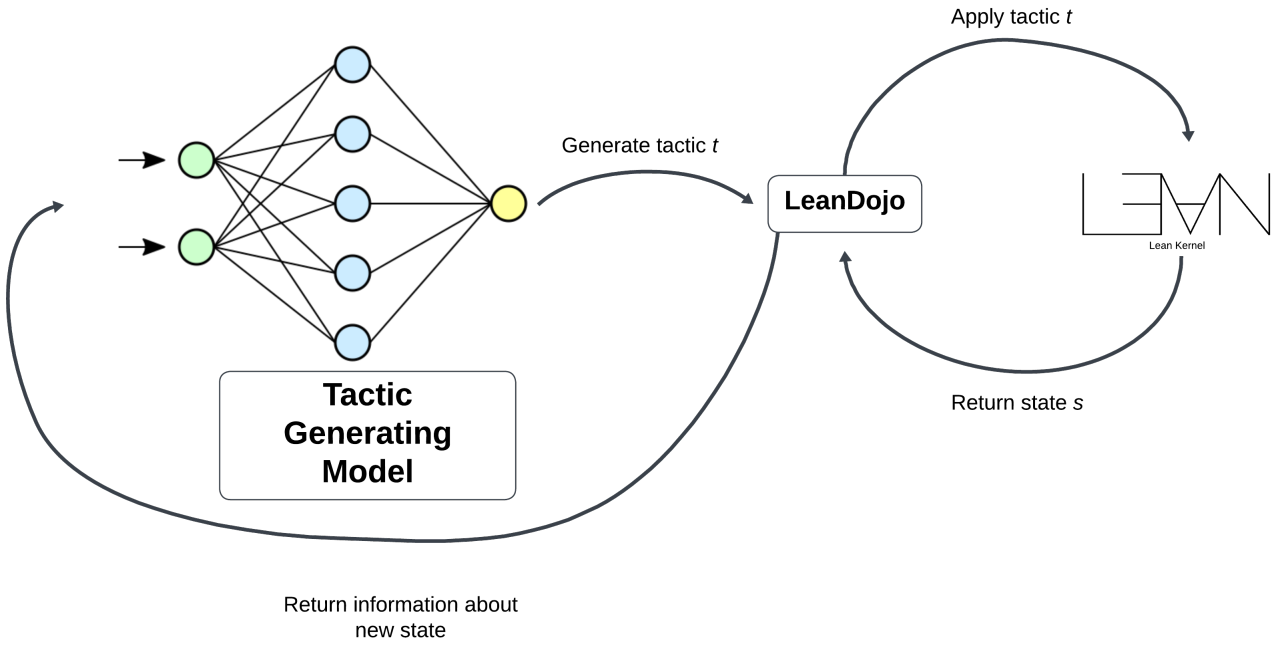


Figure 3.1: Flow of data between models and Lean

This chapter discusses the various design decisions taken in developing the theorem proving environment, as well as the tools used, before discussing the implementation of algorithms used in experiments. First, the decisions behind technologies used, especially in the case of Lean, LeanDojo, and ReProver, are discussed. It also explains slight modifications made in adapting the environment to work with reinforcement learning methods. Finally, it will examine the RL models that were designed and implemented, both of which are novel designs in the field of ATP.

3.1 Overall Architecture

The models implemented in this thesis require multiple separate systems interacting with one another in order to operate. This project deals with four separate systems that interact with one another.

- **Lean4** is the proof assistant used to verify proofs, which has the added capabilities of using theorems from Mathlib4 to assist in verification of proofs.
- **LeanDojo** is a framework and benchmark for interacting with Lean4 through the Python programming language, a popular language for machine learning.
- **Large Language Models** are a family of models used for generating tactics and embedding states. Pretrained open-source LLMs were used in combionation with the reinforcement learning-based state-value model in order to answer this thesis' first research question of whether RL methods could assist in the planning capabilities of LLM-based automated theorem provers.
- **Reinforcement Learning Models** are used to both assist in state evaluation, and in learning policies for generating text. These models and their capabilities are the primary focus of this thesis, and are a novel approach to tackling automated theorem proving.

3.2 Environment Design

This section deals with discussing the design decisions when creating a suitable framework for reinforcement learning with automated theorem proving. Since environments are formalized as a Markov decision process, it requires a state space, action space, reward function and a transition function. Since theorem proving is a deterministic process, it means that $P_a(s, s') = 1$ for some specific s' and $P_a(s, s'') = 0$ for all $s'' \neq s'$.

3.2.1 States

In the Lean theorem prover, users begin with a definition of the theorem they wish to prove. They then iteratively apply tactics which reduce this theorem into smaller components to be proved. During each step, Lean keeps track of what still needs to be proved, which it defines as "goals". These initial and intermediate goals containing information on what needs to be proven can be used as states in the environment. When a proof is complete and holds, Lean will give the feedback of "no goals", which is the definition of a terminal state in the environment.
Since reinforcement learning algorithms cannot work directly with text data, the states must

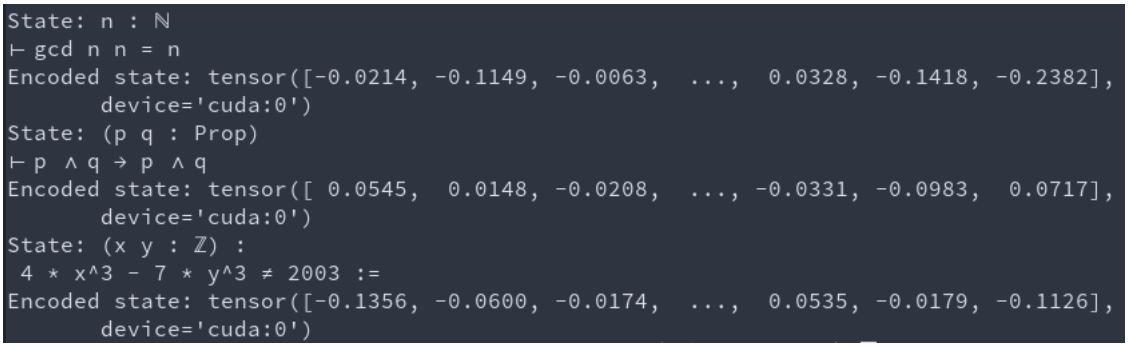


Figure 3.2: Example of embedded Lean states

be encoded into feature vectors that are embedded by a model that learns how to effectively embed the states in a meaningful way. In the environment, a pretrained, open-source language model based on the ByT5 [29] transformer architecture is used. This model, ReProver [5], is trained on the mathlib4 [4] repository with 500 million parameters. This model contains an embedding layer that has learned how to embed similar states together in the latent space, which allows the reinforcement learning models to learn state-value functions and policies without needing to additionally learn how to efficiently embed the text data.

3.2.2 Action Space

When proving theorems in Lean, users repeatedly apply tactics, which reduce the theorem until all subgoals have been proven. These tactics range from being simple in nature, such as `rfl` which checks if both sides of an equality are, in fact, equal.

```
rfl 5=5 —— true
rfl 2*2=4 —— true
rfl 0=1 —— fails
```

Other tactics are more powerful and complex, which makes the problem of automated theorem proving more difficult to solve. The tactic `intro` is used to define a hypothesis, which can theoretically be anything, leading to an infinite action space just using the `intro` keyword. Another powerful tactic is apply, which applies a proved theorem to the current state. With a repository like mathlib4, there are 100,000 potential theorems that can easily be applied. Even with this, there is no guarantee that a theorem in mathlib4 could help solve the theorem (consider unsolved problems). This can mean that within a proof, other theorems may need to be defined and proved before solving the original proof. An example of this was Andrew Wiles needing to prove the modularity theorem for elliptic curves in order to prove Fermat's Last Theorem. [30]

These tactics correspond to actions in the environment, as a continuous action space. Since

the actions are text outputs fed into Lean, there are technically an infinite number of actions that can be applied. Given that the outputs are text, any RL models trained for outputting tactics must be trained to generate text, and be based on methods that work in continuous action spaces, such as policy gradient methods.

3.2.3 Reward Function

In a reinforcement learning environment the reward is what is used by the model to learn, as rewards provide a positive or negative signal to the model so that it can be guided to take the best action in each state. Choosing a suitable reward function is an important task when designing an environment, as seen in [31], where the bicycle learned that the positive reward from turning right outweighed the negative reward from falling over, since this was easier to reach than the final goal, which took longer planning and had delayed rewards.
For this environment, there are multiple options for designing a reward signal, each with its own pros and cons. Here discuss some potential options are discussed, before noting the final design choice.
The first approach is to reward the agent for every step taken in the environment $r_{step} = \alpha$, and then to provide a large reward once a theorem has been proven. Issues arise with this, however, as the agent can learn that longer proofs generate bigger rewards, and are therefore better, something which is generally not true in theorem proving.
Another approach is to give a small negative reward to the agent at every step $r_{step} = -\alpha$. In theory this would teach the agent that solving a proof in fewer steps is better. This fails, however, as the agent simply causes the program to terminate early by applying bad tactics, in order to minimize the negative reward.
The reward function eventually chosen was to reward the agent only at the final state, and a reward of 0 for all other steps. By only rewarding the agent at the end, the agent can itself learn the values of states given how far they were from the goal within the trajectory the agent took, and states can become more highly valued if a shorter proof is ever found by the agent. This reward was modified slightly when training the tactic generator to include a negative reward for nonsensical tactics, in order to steer the agent towards better text generation.

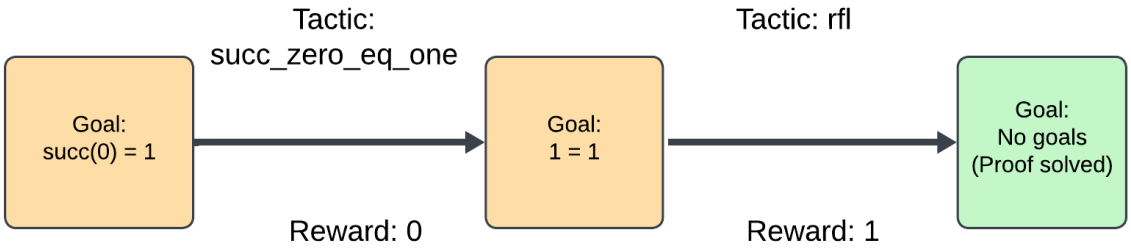


Figure 3.3: Example of reward function for a proved theorem

3.3 Large Language Models for Automated Theorem Proving

Large Language Models based on the transformer architecture currently achieve the greatest success in Automated Theorem Proving benchmarks, as they can reliably generate valid proof steps, and the next-token prediction methods employed for generating text mirror both the reinforcement learning and proof search processes of choosing the best action to taken given a certain state (the previous text in this context). The issue with LLMs is that there are no internal mechanisms for long-term future planning [32], as well as the fact that their statistical nature and large data training methods often cause the model to "hallucinate" [33], where the model relays false information due to similarity between words. These issues combined are of serious concern in using LLMs as theorem provers, since they can theoretically only generate short proofs, or can easily generate invalid theorems or search along long search trees without making good progress. Even so, generating text is a vital aspect of machine learning for theorem proving, and so some models which are incorporated to work alongside the Reinforcement Learning models in order provide better capabilities than any single model are presented.

3.3.1 ReProver

ReProver [5] is a family of Language Models trained for text generation, text embedding, and text generation with retrieval [34]. All these models are finetuned from ByT5-small, which is based on the ByT5 [29] architecture. In particular, the text embedding model is heavily used for encoding Lean states from text into feature vectors that the reinforcement learning models can then use as input.

3.3.2 Encoding text

The transformer architecture consists of an encoder-decoder structure, where the encoder projects the text data into a latent space for the decoder to sample from to generate new text, such as translated text or a continuation of the text written so far. The encoder therefore must learn to effectively represent text into a vector that compresses the text while preserving important features of the original data, such as structure, meaning, semantics, and much more.
Thanks to the effectiveness of LLM encoding models, Reinforcement Learning models can be trained by sampling from this latent space generated by the encoders, which reduces the complexity of the model as it does not need to learn embeddings itself.

3.4 Interacting with Lean

Reinforcement Learning requires interaction between the agent and the environment, with a state-action-reward-next state cycle, while also providing an observation and a state space. Lean does not provide this environment directly, as it only takes a theorem and tactics, and returns the current state of the proof. Interfacing between Python and Lean is another important factor for this thesis, as Python is the primary language used in machine learning, and has support and infrastructure for designing and implementing models.

3.4.1 LeanDojo

LeanDojo [5] is a Python framework for interfacing between machine learning models and Lean, and includes a suite of datasets and benchmarks for training and testing models. LeanDojo provides an interface for extracting data from GitHub repositories containing Lean code, by tracing the repositories and creating abstract syntax trees (ASTs) from proofs within these repositories. ASTs are able to contain information about the structure and flow of proofs, which is important when training machine learning models as applying tactics is not a commutative operator, i.e tactics must be applied in a specific order to create a valid proof.

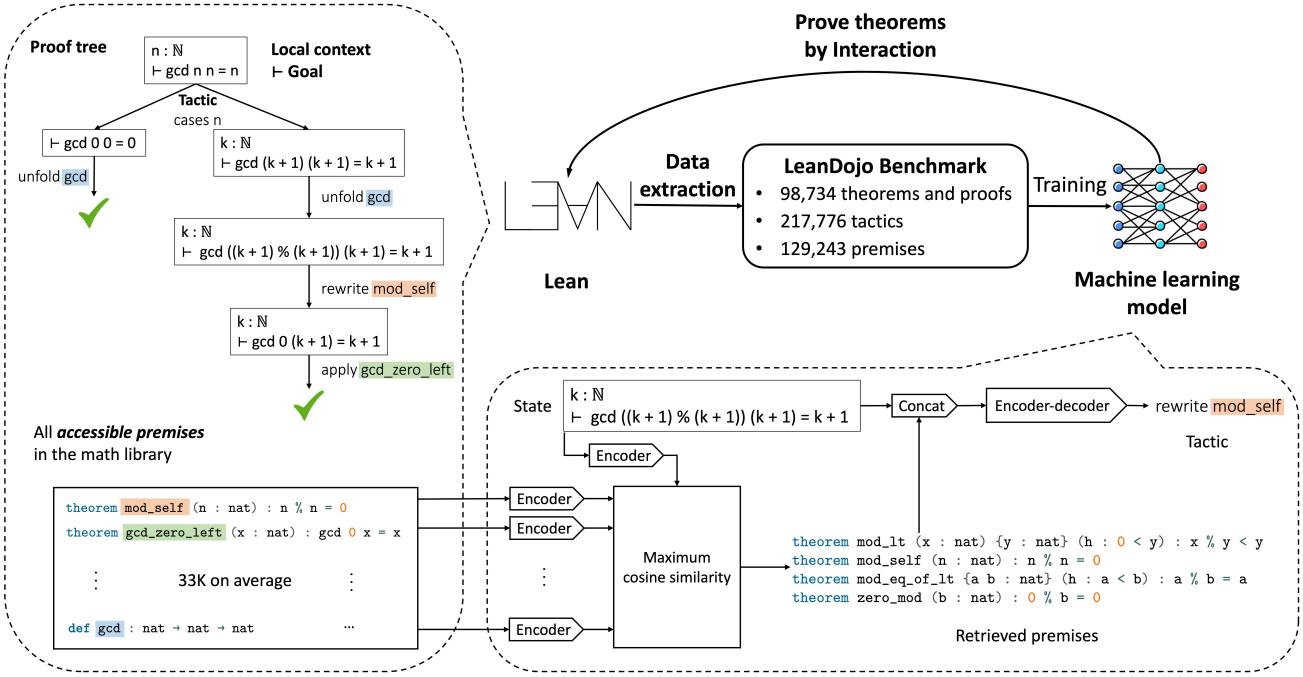


Figure 3.4: Example of training a model with that interacts with Lean through LeanDojo

The datasets and benchmarks offered by LeanDojo are generated from MathLib 4, which consists of over 140,000 theorems handwritten by human experts. This provides high-quality training data that mimicks how humans write proofs, while also being valid and correct

proofs thanks to verification from the Lean kernel.

3.5 Reinforcement Learning Models

Herethe specifications of the designs and implementations for the state-value model for evaluating proof states, as well as the policy gradient model for generating tactics are discussed.

3.5.1 State-Value Model

The state-value model is a deep neural network designed using a multilayer perceptron, which uses an encoded state as its input, and a single real-valued output which indicates the learned value of the current state. This model is a novel contribution to the field of ATP, designed and implemented for the purposes of this thesis.
The state-value model consists of a pretrained Encoder block which embeds the text data into a 1472-dimension vector, which is used as the input to the MLP model. This model uses dense layers in a feedforward network with hidden states of size 1024 and 512 respectively, and output size of 1 to predict a value for the state. Each The model uses the ReLU activation function. The loss function to be optimized is calculated using Temporal-Difference (TD) Learning. The reward is a 1.0 for transitioning into a terminal state, which accounts for a "proof solved", and a 0 reward for all other state transitions. When optimizing the Bellman Equation, this then allows the model to allocate values to states which are indicative of how close to the solution they are during training. The model is updated using target networks, which keep track of two networks, and only update the parameters every few batches in order to improve stability. The Adam [35] optimizer is used to train the model during backpropagation, with a learning rate of 5e-4, batch size of 128, and a decay parameter γ of 0.99. The goal of the state-value model is to explore how RL methods can be combined with LLMs in order to improve the capabilities of LLMs in searching and planning, as laid out in research question 1. All parameters are shown in Appendix A1.1

3.5.2 Tactic Generator Policy Gradient Model

The tactic generating model is a DNN model that outputs a fixed-length series of tokens which are Lean tactics required to solve the proof. This model is used to interact directly with Lean in order to solve proofs without the need for a language model. The Tactic Generator model is structured in a similar manner to the State-Value model in its initial layers, but its output layers are of size 50 instead of 1, where each "action" corresponds to a token. This means that each output neuron should be an integer between 0 and 384, which corresponds to the number of unique tokens in the dataset (e.g 'a', 'B', 'λ'). This is a new

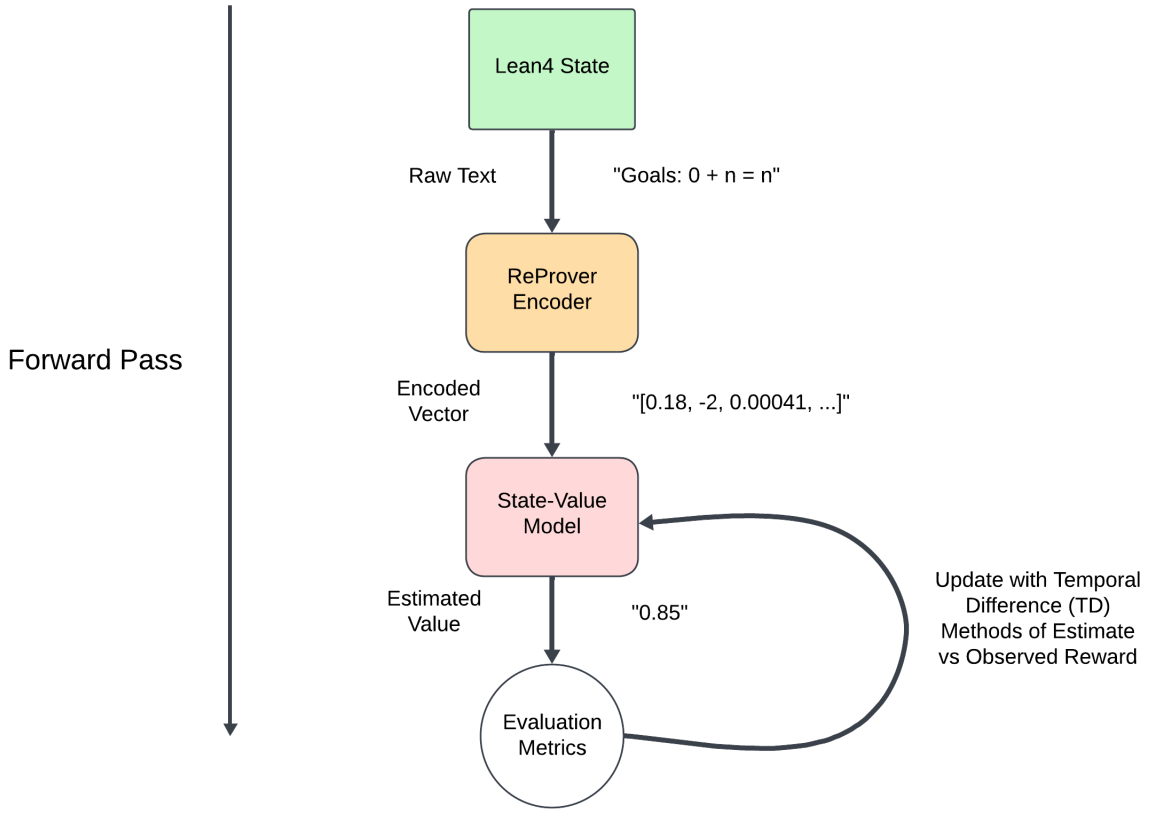


Figure 3.5: Example of training pass through the state-value model

approach to training text generation models, where RL methods were used in order to bridge the gap between models that can and cannot plan, i.e RL models and language generation models.

For this model the reward was the same as before, i.e 0 for regular state transition and 1 for terminal state of proof solved. A negative reward of -1 was also implemented for text that failed in Lean, which was to guide the model to generate valid text, and not random output. To optimize the model against the reward Proximal Policy Optimization (PPO) [21] was used. This model consisted of 2 hidden layers of size 256 and 128 before the final layer of size 50, with the ReLU transfer function. The same hyperparameters of Adam, batch size, learning rate and decay parameter were used in this model as in the State-Value model. This model is used to directly relate to research questions 2 and 3, which explore the capabilities of RL models without using tactics generated by LLMs while proving theorems. The reasoning for once again using the ReProver embedding model is for simplicity. It is entirely possible to train the RL model to also learn to embed states, but this adds extra complexion to a model, would slow the rate of learning, and causes difficulties in debugging the model. All parameters for the tactic generator model are shown in Appendix A1.2

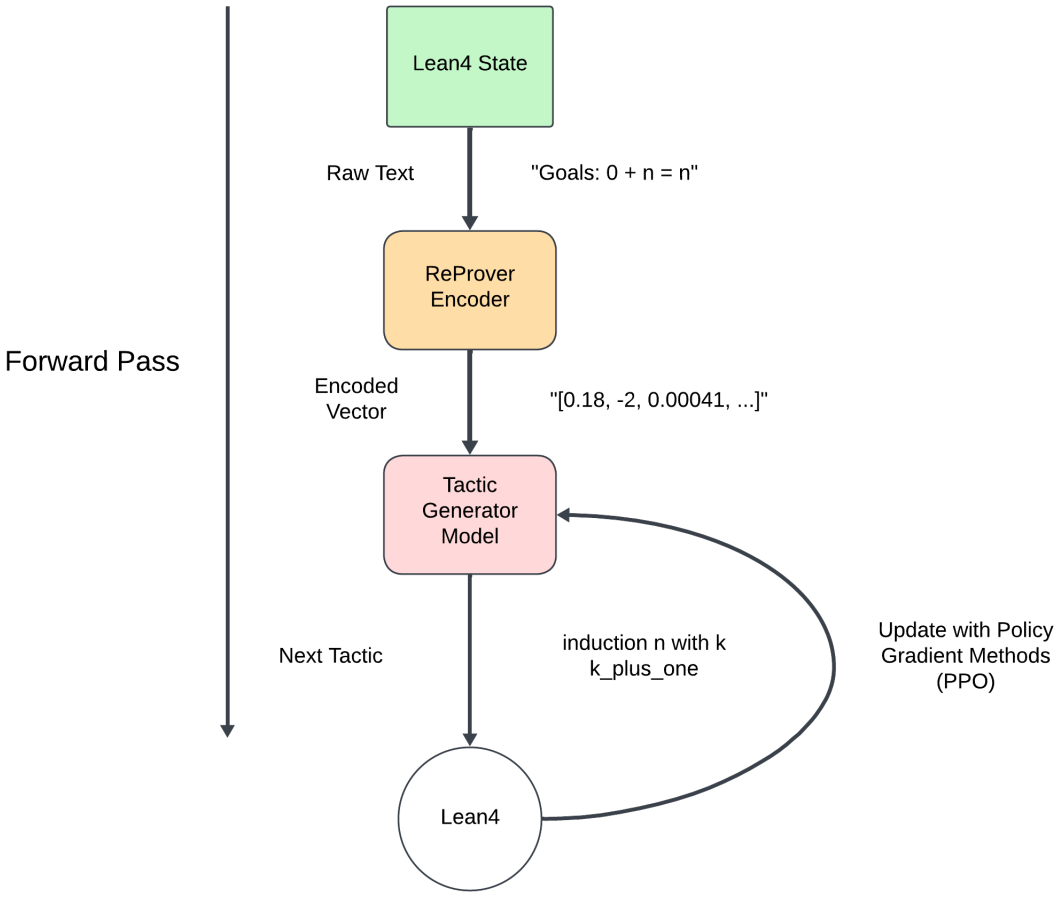


Figure 3.6: Architecture of the text generating model

4 Evaluation

This section discusses the performance of Reinforcement Learning algorithms on various benchmarks for Automated Theorem Proving, as well as limitations and analysis of RL in the field of ATP, compared with current State-of-the-art methods.

4.1 LeanDojo Benchmark

The LeanDojo framework includes three datasets created from the Mathlib repository, for training, testing, and performing validation on trained models. The test dataset is used for evaluating models, with metrics being based on how many of the problems in the dataset resulted in a solved proof. The LeanDojo benchmark is split in a 98,734-2000-2000 manner The benchmark actually consists of 2 different methods of splitting these datasets, with the Random Split being an arbitrary split of Mathlib, and the Novel Premises reserving more difficult problems for the test and validation sets. Both benchmarks contain the same data but are structured differently.

4.1.1 Random Split

All models were trained on the random split version of the LeanDojo benchmark, and thus evaluations were run only on this split, as there could be contamination in the train dataset for the novel premises test split. Evaluation on the test data is performed by running a proof search, where the score of a model is evaluated by the number of theorems it proves. A proof fails if either a hard timeout limit is reached during the proof search, or Lean crashes during the proof, which can happen due to invalid inputs into the Lean kernel. For testing, a timeout of 60 seconds per proof was selected.
The test dataset contains a series of 2000 theorems, but due to computational limits crashes would occur after 85 proofs due to proof trees becoming too large for the testing machine to

Dataset	Train	Validation	Test
Size	98,734	2,000	2,000

Table 4.1: Size of LeanDojo dataset splits

Model	Attempted	Proved	#Nodes searched	Avg nodes searched
ReProver	85	31	79	2.55
ReProver with State-Value model	85	29	45	1.55

Table 4.2: Comparison between baseline ReProver and state-value augmented search

handle. Tests were run for baseline ReProver which chooses states to expand using beam search against ReProver modified with the state-value function for choosing states to evaluate. It should also be noted that due to this additional network, each node searched by the state-value model takes slightly longer to evaluate, and therefore the timeout can be reached with fewer nodes actually searched.
On a subset of the test dataset consisting of 85 proofs, the baseline ReProver model proved 31 theorems compared to 29 theorems proved by the state-value modifications. In terms of search efficiency, however, the state-value model was able to prove these theorems by searching 45 nodes compared to the 79 nodes searched by baseline ReProver, which shows an average of 1.55 nodes searched by the baseline, compared to the 2.55 searched by ReProver. A majority of proofs were also solved in a single step (such as with `rfl`), with 22 of proofs by each being solved in this single-step manner where search was not required. That means that on those shared 7 proofs, baseline ReProver runs search over 6.33 nodes compared to only 3.29 searched by state value models on average. For the two theorems that baseline ReProver proved that the state-value function did not prove, state-value function evaluated the best states as too low and therefore they were not fully expanded.

Model	Attempted	Proved	#Nodes searched	Avg nodes searched
ReProver	85	9	57	6.33
ReProver with State-Value model	85	7	23	3.29

Table 4.3: Comparison between baseline ReProver and state-value augmented search when adjusted for single-step proofs

4.2 Analysis of performance

The action generator policy gradient model performs poorly on all datasets, due to the inherent difficulties of generating text using pure multilayered perceptron networks. The issues arise from the fact that text requires positional information, in order to produce legible results and multilayered perceptrons do not support this kind of operation. RNNs and Transformers are the better option for the problem of producing text, and using RL models to produce hidden states for these models was an approach that was briefly explored but was unable to produce results due to difficulties in learning hidden states for pretrained models. Training models that learn in an end-to-end fashion, which would train both the RL model and a RNN or Transformer model simultaneously is a future apporach that could yield

impressive results.
This result shows a disappointing answer to research questions 2 and 3, in regards to using purely RL-based methods for theorem proving applications. These results reiterate the importance of language models in this field, and help to cement the place of LLMs in the field of theorem proving.
The State-Value model performed much better in this regard, as it was able to optimize the search paths taken by the model, and did not have the additional burden of learning to generate tactics, as the ReProver model acted as the action generator. While the LeanDojo benchmark only uses number of theorems proved as an evaluation metric, reducing search time is an important factor in theorem proving and should be seriously considered as another metric of interest for a theorem prover. In this case, the state-value model was able to improve the search speed in some cases for proof search, with regular log probability queue searching 2.55 states on average and state-value queue searching 1.55 states on average, which increases to 6.33 and 3.29 nodes searched respectively, when multi-step proofs are required.

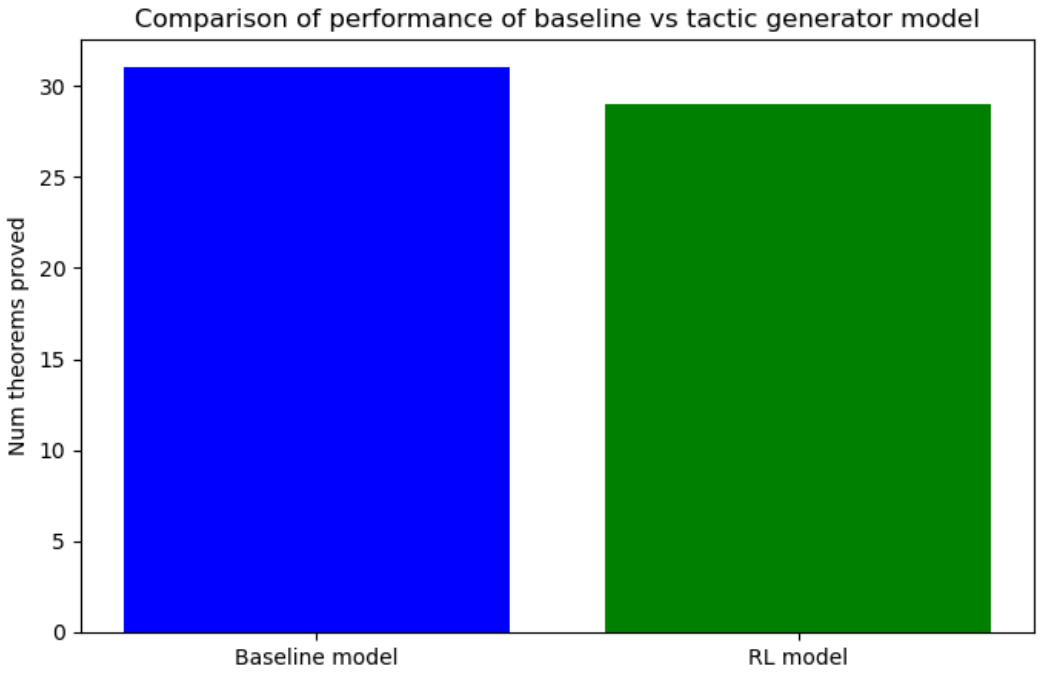


Figure 4.1: Performance comparison on number of theorems proved

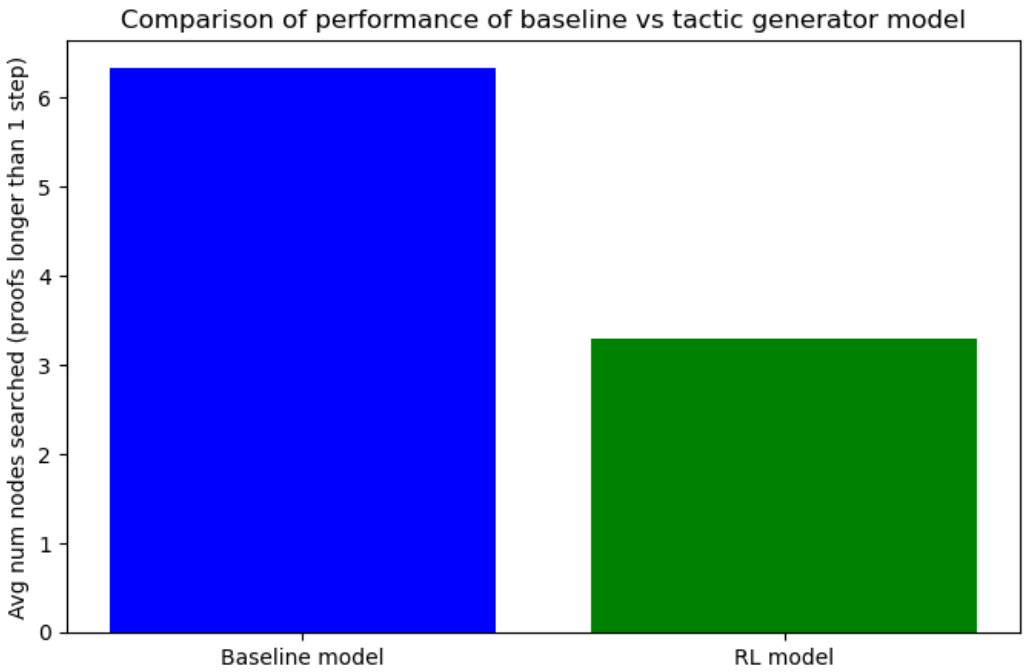


Figure 4.2: Performance comparison on search (lower is better)

4.3 Limitations and critiques of RL methods

While it has been shown that RL models have proven capable of improving search within the field of ATP, it is important to note some limitations and critiques of these augmentations to the language models.

4.3.1 Training additional models

In order to use RL models that can assist with the proof search aspect of ATP, these models first need to be trained. It must be noted that training AI models is having a significant negative impact on the environment [36], and the fact that are training additional models being trained just to assist existing models is a pattern that cannot work in the long term. However, the scale and environmental impact of a RL model for this task is negligible compared to the compute required to train the original language models for generating the Lean code, on a scale of about 1:1000, so perhaps future work could focus on training smaller generative models that make use of small state-value models that overall use less compute than the traditional larger generative models.

4.3.2 Reliance on LLMs

As noted in the implementation section, the state-value model uses embeddings derived from the ReProver model in order to hasten the training process for the state-value model and remove the need for this model to learn embeddings from text, and instead work directly with the embedding vectors produced by the ReProver model. This approach, while a useful method in training a model compatible with ReProver, lacks generalization abilities and would need to either always use the specific embedding model it was trained on for embedding states, or would need to be trained with a new embedding model. This also causes issues with the fact that, if the ReProver embedding model needs to be updated, such as after new theorems have been added to Mathlib, then once again the state-value model would most likely need to be retrained as well.

5 Conclusion

The main findings of this thesis are discussed in this chapter, along with their relations to the research questions. Remaining research questions and new questions raised throughout the thesis are then discussed.

5.1 Thesis Contributions

Chapter 1 discussed the importance of proving mathematical theorems the benefits that computers can bring to the field. It examined the great lengths that people have gone to in order to formalize a broad range of mathematical topics in Lean, and the effort that has gone into building a repository containing these in Mathlib.
Chapter 2 explains the background knowledge required for understanding both the field of theorem proving, and the technologies involed in deep reinforcment learning and language models, both of which are heavily used in this thesis in order to create models that can prove complex theorems.
The frameworks described in Chapter 2 were built upon by discussing specific implementation and design choices made throughout this thesis in Chapter 3. Here it was discussed exactly how the environment for theorem proving with RL was designed, and the specific technologies that were implemented in order to create models capable of proving theorems through tactic generation and proof search.
Chapter 4 takes the models described in Chapter 3 and evaluates them against current state of the art on the LeanDojo benchmark, a standardized dataset in the field of ATP. A comparison was performed on strengths and weaknesses these RL-based models bring in regards to the current models being used, and finally it discusses the drawbacks and limitations to training additional models for state evaluation.

5.2 Future Work

Various potential ideas and applications arose from the results of this thesis, that were not implemented either due to a variety of factors, such as being outside the scope of the thesis,

specific ideas being too complex to implement and limited compute and/or time. A list of these is seen below

- **State-Value models for text generation:** The fact that RL methods were able to improve search capabilities in terms of speed for theorem proving leads one to wonder if this method can be used to allow for general text generation models to better plan. This could lead to improved capabilities in language models in a variety of tasks, but the lack of environment to test this in is a significant limiting factor
- **RL methods for exploring latent space of LLMs:** This thesis used the encoder structure of LLMs in order to embed text for RL models to use, but a potential approach is to have the RL agent modify the latent space to guide the LLM in a specific direction. Some experiments were performed to test the validity, but complexity in implementing this method along with lack of support to directly modify the latent space of pretrained models meant that this approach was put aside to focus on the methods used in the thesis.
- **Exploration of latent space:** One important part of the success of the state-value model is the need for a good embedding model. While the embedding model is quite strong, it is unknown exactly how theorems embedded by this model relate to the embeddings of other theorems used within the model itself. Methods such as GloVe [37] have shown that text embedding models trained in specific ways can learn vector representations that have meaningful relations to similar words. One future approach could be to train an embedding model that embeds theorems as a product of the tactics used when proving the theorem, and then training a RL model to search for paths between these vector representations. This is similar to future work 2 but without necessarily relying on LLMs.
- **Explore how this approach works with other models:** Naturally next steps to take would be to explore if the results shown in this thesis also hold for other models, especially larger ones. The main reason for this not being tested in this thesis is the lack of open-source models that are trained purely to produce Lean code.

Bibliography

[1] K. Appel and W. Haken, "The solution of the four-color-map problem," *Scientific American*, vol. 237, no. 4, pp. 108–121, 1977. [Online]. Available: http://www.jstor.org/stable/24953967

[2] N. G. de Bruijn, "The mathematical language automath, its usage, and some of its extensions," in *Symposium on Automatic Demonstration*, M. Laudet, D. Lacombe, L. Nolin, and M. Schützenberger, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 1970, pp. 29–61.

[3] L. de Moura, S. Kong, J. Avigad, F. van Doorn, and J. von Raumer, "The lean theorem prover (system description)," in *Automated Deduction - CADE-25*, A. P. Felty and A. Middeldorp, Eds. Cham: Springer International Publishing, 2015, pp. 378–388.

[4] T. mathlib Community, "The lean mathematical library," in *Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs*, ser. CPP 2020. New York, NY, USA: Association for Computing Machinery, 2020, p. 367–381. [Online]. Available: https://doi.org/10.1145/3372885.3373824

[5] K. Yang, A. M. Swope, A. Gu, R. Chalamala, P. Song, S. Yu, S. Godil, R. Prenger, and A. Anandkumar, "Leandojo: Theorem proving with retrieval-augmented language models," 2023.

[6] N. D. Megill, *Metamath: A Computer Language for Mathematical Proofs*. Morrisville, North Carolina: Lulu Press, 2019, `http://us.metamath.org/downloads/metamath.pdf`.

[7] Y. Bertot and P. Castéran, *Interactive theorem proving and program development: Coq'Art: the calculus of inductive constructions*. Springer Science & Business Media, 2013.

[8] A. M. Dakhel, V. Majdinasab, A. Nikanjam, F. Khomh, M. C. Desmarais, Z. Ming, and Jiang, "Github copilot ai pair programmer: Asset or liability?" 2023.

[9] S. Polu and I. Sutskever, "Generative language modeling for automated theorem proving," 2020.

[10] T. Nipkow, L. C. Paulson, and M. Wenzel, *Isabelle/HOL: a proof assistant for higher-order logic*. Springer Science & Business Media, 2002, vol. 2283.

[11] L. Kovács and A. Voronkov, "First-order theorem proving and vampire," in *Computer Aided Verification*, N. Sharygina and H. Veith, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 1–35.

[12] A. Radford, K. Narasimhan, T. Salimans, and I. Sutskever, "Improving language understanding by generative pre-training," 2018.

[13] L. Blaauwbroek, J. Urban, and H. Geuvers, "Tactic learning and proving for the coq proof assistant," in *LPAR23. LPAR-23: 23rd International Conference on Logic for Programming, Artificial Intelligence and Reasoning*, ser. EPiC Series in Computing, E. Albert and L. Kovacs, Eds., vol. 73. EasyChair, 2020, pp. 138–150. [Online]. Available: https://easychair.org/publications/paper/JLdB

[14] J. Rute, M. Olšák, L. Blaauwbroek, F. I. S. Massolo, J. Piepenbrock, and V. Pestun, "Graph2tac: Learning hierarchical representations of math concepts in theorem proving," 2024.

[15] I. J. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning*. Cambridge, MA, USA: MIT Press, 2016, http://www.deeplearningbook.org.

[16] F. Rosenblatt, "The perceptron: A probabilistic model for information storage and organization in the brain." *Psychological Review*, vol. 65, no. 6, pp. 386–408, 1958. [Online]. Available: http://dx.doi.org/10.1037/h0042519

[17] R. S. Sutton and A. G. Barto, *Reinforcement learning: An introduction*. MIT press, 2018.

[18] R. S. Sutton, "Learning to predict by the methods of temporal differences," *Mach. Learn.*, vol. 3, no. 1, p. 9–44, aug 1988. [Online]. Available: https://doi.org/10.1023/A:1022633531479

[19] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. Riedmiller, "Playing atari with deep reinforcement learning," 2013.

[20] D. Silver, A. Huang, C. J. Maddison, A. Guez, L. Sifre, G. van den Driessche, J. Schrittwieser, I. Antonoglou, V. Panneershelvam, M. Lanctot, S. Dieleman, D. Grewe, J. Nham, N. Kalchbrenner, I. Sutskever, T. Lillicrap, M. Leach, K. Kavukcuoglu, T. Graepel, and D. Hassabis, "Mastering the game of go with deep

neural networks and tree search," *Nature*, vol. 529, pp. 484–503, 2016. [Online]. Available: http://www.nature.com/nature/journal/v529/n7587/full/nature16961.html

[21] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov, "Proximal policy optimization algorithms," 2017.

[22] R. S. Sutton, D. McAllester, S. Singh, and Y. Mansour, "Policy gradient methods for reinforcement learning with function approximation," in *Advances in Neural Information Processing Systems*, S. Solla, T. Leen, and K. Müller, Eds., vol. 12. MIT Press, 1999. [Online]. Available: https://proceedings.neurips.cc/paper_files/paper/1999/file/464d828b85b0bed98e80ade0a5c43b0f-Paper.pdf

[23] V. Konda and J. Tsitsiklis, "Actor-critic algorithms," in *Advances in Neural Information Processing Systems*, S. Solla, T. Leen, and K. Müller, Eds., vol. 12. MIT Press, 1999. [Online]. Available: https://proceedings.neurips.cc/paper_files/paper/1999/file/6449f44a102fde848669bdd9eb6b76fa-Paper.pdf

[24] J. Schulman, S. Levine, P. Moritz, M. I. Jordan, and P. Abbeel, "Trust region policy optimization," 2017.

[25] M. Freitag and Y. Al-Onaizan, "Beam search strategies for neural machine translation," in *Proceedings of the First Workshop on Neural Machine Translation*. Association for Computational Linguistics, 2017. [Online]. Available: http://dx.doi.org/10.18653/v1/W17-3207

[26] D. E. Rumelhart, G. E. Hinton, and R. J. Williams, *Learning internal representations by error propagation*. Cambridge, MA, USA: MIT Press, 1986, p. 318–362.

[27] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser, and I. Polosukhin, "Attention is all you need," 2023.

[28] D. Bahdanau, K. Cho, and Y. Bengio, "Neural machine translation by jointly learning to align and translate," 2016.

[29] L. Xue, A. Barua, N. Constant, R. Al-Rfou, S. Narang, M. Kale, A. Roberts, and C. Raffel, "Byt5: Towards a token-free future with pre-trained byte-to-byte models," 2022.

[30] A. Wiles, "Modular elliptic curves and fermat's last theorem," *Annals of Mathematics*, vol. 141, no. 3, pp. 443–551, 1995. [Online]. Available: http://www.jstor.org/stable/2118559

[31] J. Randløv and P. Alstrøm, "Learning to drive a bicycle using reinforcement learning and shaping," in *International Conference on Machine Learning*, 1998. [Online]. Available: https://api.semanticscholar.org/CorpusID:28257125

[32] K. Valmeekam, A. Olmo, S. Sreedharan, and S. Kambhampati, "Large language models still can't plan (a benchmark for LLMs on planning and reasoning about change)," in *NeurIPS 2022 Foundation Models for Decision Making Workshop*, 2022. [Online]. Available: https://openreview.net/forum?id=wUU-7XTL5XO

[33] N. M. Guerreiro, E. Voita, and A. F. T. Martins, "Looking for a needle in a haystack: A comprehensive study of hallucinations in neural machine translation," 2023.

[34] V. Karpukhin, B. Oğuz, S. Min, P. Lewis, L. Wu, S. Edunov, D. Chen, and W. tau Yih, "Dense passage retrieval for open-domain question answering," 2020.

[35] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," 2017.

[36] OECD, "Measuring the environmental impacts of artificial intelligence compute and applications," no. 341, 2022. [Online]. Available: https://www.oecd-ilibrary.org/content/paper/7babf571-en

[37] J. Pennington, R. Socher, and C. D. Manning, "Glove: Global vectors for word representation," in *Empirical Methods in Natural Language Processing (EMNLP)*, 2014, pp. 1532–1543. [Online]. Available: http://www.aclweb.org/anthology/D14-1162

A1 Appendix

A1.1 Graphs of training models

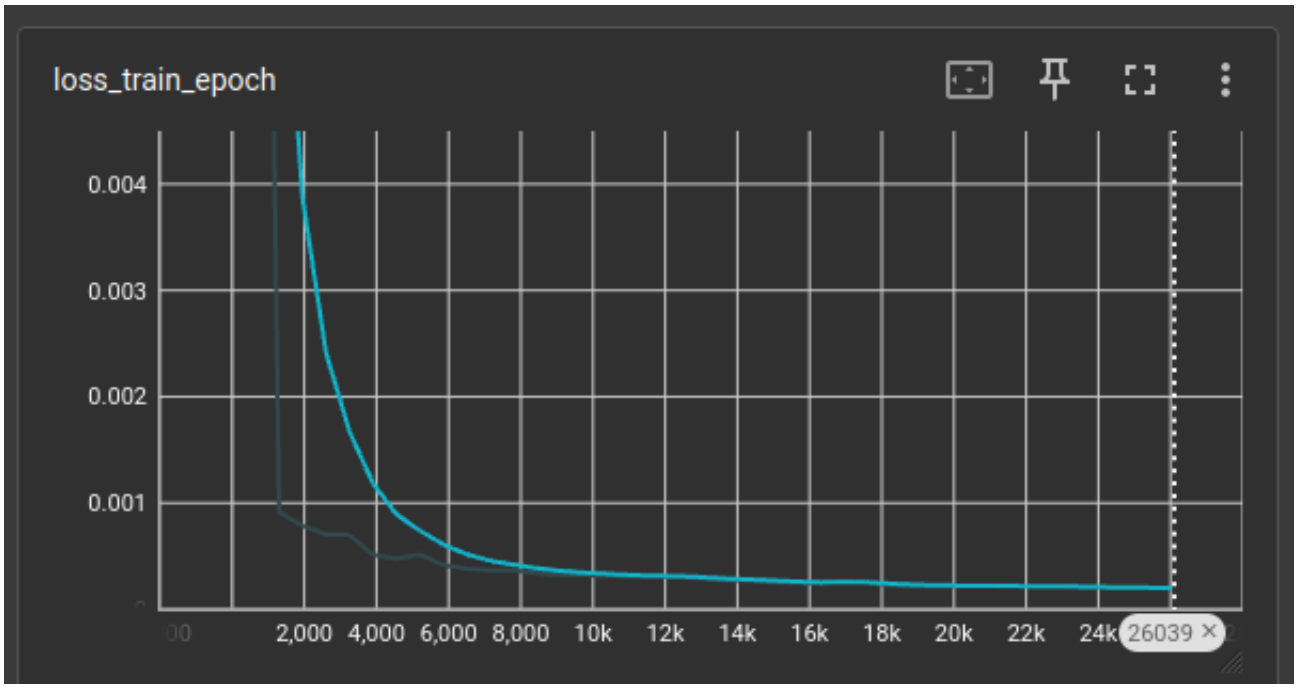


Figure A1.1: Loss curve for training of the temporal difference model

A1.2 Model Parameters

A1.2.1 Parameters for State-Value Model

Parameter	Value
Total Steps	26000
Reward Range	0, 1
Decay	0.99
Learning Rate	5e-4
Batch size	128
Optimizer	Adam
Input size	1472
Hidden Layer 1 size	1024
Hidden Layer 2 size	512
Output size	1

Table A1.1: State-Value model Parameters

A1.2.2 Parameters for Tactic Generator Model

Parameter	Value
Total Steps	40000
Reward Range	-1, 0, 1
Decay	0.99
Learning Rate	5e-4
Batch size	128
Optimizer	Adam
Input size	1472
Hidden Layer 1 size	1024
Hidden Layer 2 size	512
Output size	50

Table A1.2: State-Value model Parameters

A1.3 Example of LeanDojo datapoint


```
[{"url": "https://github.com/leanprover-community/mathlib4",
"commit": "3ce43c18f614b76e161f911b75a3e1ef641620ff", "file_path":
"Mathlib/Topology/Algebra/Monoid.lean", "full_name":
"exists_open_nhds_one_split", "start": [475, 1], "end":
[479, 61], "traced_tactics": [{"tactic":
```

"have : (fun a : M \u00d7 M => a.1 * a.2) \u207b\u00b9'
s \u2208 \ud835\udcdd ((1, 1) : M \u00d7 M) :=\n tendsto_mul
(by simpa only [one_mul] using hs)", "annotated_tactic": ["have
: (fun a : M \u00d7 M => a.1 * a.2) \u207b\u00b9' s
\u2208 \ud835\udcdd ((1, 1) : M \u00d7 M) :=\n
<a>tendsto_mul (by simpa only [<a>one_mul]
using hs)", [{"full_name": "tendsto_mul",
"def_path": "Mathlib/Topology/Algebra/Monoid.lean",
"def_pos": [113, 9], "def_end_pos": [113, 20]}, {"full_name":
"one_mul", "def_path": "Mathlib/Algebra/Group/Defs.lean", "def_pos":
[464, 9], "def_end_pos": [464, 16]}]], "state_before": "\u03b9 : Type
u_1\n\u03b1 : Type u_2\nX : Type u_3\nM : Type u_4\nN : Type
u_5\ninst\u271d\u00b3 : TopologicalSpace X\ninst\u271d\u00b2 :
TopologicalSpace M\ninst\u271d\u00b9 : Monoid M\ninst\u271d :
ContinuousMul M\ns : Set M\nhs : s \u2208 \ud835\udcdd 1\n\u22a2
\u2203 V, IsOpen V \u2227 1 \u2208 V \u2227 \u2200 (v : M), v
\u2208 V \u2192 \u2200 (w : M), w \u2208 V \u2192 v * w \u2208 s",
"state_after": "\u03b9 : Type u_1\n\u03b1 : Type u_2\nX : Type u_3\nM :
Type u_4\nN : Type u_5\ninst\u271d\u00b3 : TopologicalSpace
X\ninst\u271d\u00b2 : TopologicalSpace M\ninst\u271d\u00b9 :
Monoid M\ninst\u271d : ContinuousMul M\ns : Set M\nhs : s \u2208
\ud835\udcdd 1\nthis : (fun a => a.1 * a.2) \u207b\u00b9' s \u2208
\ud835\udcdd (1, 1)\n\u22a2 \u2203 V, IsOpen V \u2227 1 \u2208 V
\u2227 \u2200 (v : M), v \u2208 V \u2192 \u2200 (w : M), w \u2208 V
\u2192 v * w \u2208 s"}, {"tactic": "simpa only [prod_subset_iff]
using exists_nhds_square this", "annotated_tactic": ["simpa only
[<a>prod_subset_iff] using <a>exists_nhds_square this",
[{"full_name": "Set.prod_subset_iff", "def_path":
"Mathlib/Data/Set/Prod.lean", "def_pos": [100, 9], "def_end_pos":
[100, 24]}, {"full_name": "exists_nhds_square", "def_path":
"Mathlib/Topology/Constructions.lean", "def_pos": [674, 9],
"def_end_pos": [674, 27]}]], "state_before": "\u03b9 :
Type u_1\n\u03b1 : Type u_2\nX : Type u_3\nM : Type u_4\nN :
Type u_5\ninst\u271d\u00b3 : TopologicalSpace X\ninst\u271d\u00b2
: TopologicalSpace M\ninst\u271d\u00b9 : Monoid M\ninst\u271d :
ContinuousMul M\ns : Set M\nhs : s \u2208 \ud835\udcdd 1\nthis :
(fun a => a.1 * a.2) \u207b\u00b9' s \u2208 \ud835\udcdd
(1, 1)\n\u22a2 \u2203 V, IsOpen V \u2227 1 \u2208 V \u2227 \u2200
(v : M), v \u2208 V \u2192 \u2200 (w : M), w \u2208 V \u2192 v * w

\u2208 s", "state_after": "no goals"}, {"tactic": "simpa only [one_mul] using hs", "annotated_tactic": ["simpa only [<a>one_mul] using hs", [{"full_name": "one_mul", "def_path": "Mathlib/Algebra/Group/Defs.lean", "def_pos": [464, 9], "def_end_pos": [464, 16]}]], "state_before": "\u03b9 : Type u_1\n\u03b1 : Type u_2\nX : Type u_3\nM : Type u_4\nN : Type u_5\ninst\u271d\u00b3 : TopologicalSpace X\inst\u271d\u00b2 : TopologicalSpace M\ninst\u271d\u00b9 : Monoid M\inst\u271d : ContinuousMul M\ns : Set M\nhs : s \u2208 \ud835\udcdd 1\n\u22a2 s \u2208 \ud835\udcdd (1 * 1)", "state_after": "no goals"}]}