Abstract Reasoning with Deep Learning

cumulative dissertation

by

Stanisław J. Purgał

submitted to the Faculty of Mathematics, Computer Science and Physics of the University of Innsbruck

in partial fulfillment of the requirements for the degree of academic degree

advisors: Assoc.-Prof. Dr. Cezary Kaliszyk

Innsbruck, 27 January 2023

universität innsbruck

cumulative dissertation

Abstract Reasoning with Deep Learning

Stanisław J. Purgał (11849755)
sjpurgal@gmail.com

27 January 2023

advisors: Assoc.-Prof. Dr. Cezary Kaliszyk

Eidesstattliche Erklärung

Ich erkläre hiermit an Eides statt durch meine eigenhändige Unterschrift, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe. Alle Stellen, die wörtlich oder inhaltlich den angegebenen Quellen entnommen wurden, sind als solche kenntlich gemacht.
Die vorliegende Arbeit wurde bisher in gleicher oder ähnlicher Form noch nicht als Magister-/Master-/Diplomarbeit/Dissertation eingereicht.

Datum	Unterschrift

Abstract

The field of Artificial Intelligence has seen great advances with the use of Deep Neural Networks. However, the problem of creating a system capable of abstract reasoning remains unsolved.

One way to force AI to perform abstract reasoning is to directly learn an abstract task – such as Automated Theorem Proving. In ATP, the goal is to construct a formal proof of a mathematical statement. This requires finding a correct sequence of inferences in an exponentially large space of possibilities. This search can be guided by a Deep Neural Network.

Toward this end, I have worked on creating a neural architecture that would work well with mathematical formulas and provide a way to generate training data to train such neural networks.

A different way of tackling the abstract reasoning problem is to use Inductive Logic Programming. In this machine learning paradigm, the task is to construct a logic program to explain given data – which naturally tends to be abstract and general. In this field, I did work on both improving existing ILP methods and integrating ILP with Deep Learning.

Abstract in Polish

W dziedzinie sztucznej inteligencji nastąpił ogromny postęp dzięki zastosowaniu Głębokich Sieci Neuronowych. Jednak problem stworzenia systemu zdolnego do abstrakcyjnego rozumowania pozostaje nierozwiązany.

Jednym ze sposobów zmuszenia Sztucznej Inteligencji (AI) do wykonywania abstrakcyjnych rozumowań jest bezpośrednie uczenie abstrakcyjnego zadania - na przykład Automatyczne Dowodzenie Twierdzeń (ATP). W ATP, celem jest skonstruowanie formalnego dowodu twierdzenia matematycznego. Wymaga to znalezienia poprawnej sekwencji wnioskowań w wykładniczo dużej przestrzeni możliwości. Poszukiwania te mogą być kierowane przez głęboką sieć neuronową.

W tym celu pracowałem nad stworzeniem architektury neuronowej, która będzie dobrze pracować z formułami matematycznymi i nad zapewnieniem sposobu na generowanie danych treningowych do trenowania takich sieci neuronowych.

Innym sposobem rozwiązania problemu abstrakcyjnego rozumowania jest użycie Indukcyjnego Programowania w Logice (ILP). W tym paradygmacie uczenia maszynowego zadaniem jest skonstruowanie programu logicznego który pasuje do zadanych danych wejściowych – programy takie mają tendencję do bycia abstrakcyjnymi i ogólnymi. W tej dziedzinie pracowałem zarówno nad ulepszeniem istniejących metod ILP, jak i integracją ILP z Głębokimi Sieciami Neuronowymi.

Abstract in German

Der Bereich der Künstlichen Intelligenz hat durch den Einsatz von Deep Neural Networks große Fortschritte erzielt. Jedoch ein System zu konzipieren, welches in der Lage ist abstrakte Schlussfolgerungen zu ziehen, bleibt ein ungelöstes Problem.

Eine Möglichkeit, Künstlicher Intelligenz abstraktes Denken beizubringen, besteht darin abstrakte Aufgaben direkt zu erlernen, wie z.B.das automatisierte Beweisen von Theoremen (ATP). Bei ATP besteht das Ziel darin, einen formalen Beweis für eine mathematische Aussage zu konstruieren. Dazu muss eine korrekte Folge von Schlussfolgerungen in einem exponentiell großen Raum von Möglichkeiten gefunden werden. Diese Suche kann von einem tiefen neuronalen Netz geleitet werden.

Zu diesem Zweck habe ich an der Entwicklung einer neuronalen Architektur gearbeitet, welche sich auf mathematische Formeln spezialisiert. Zudem habe ich eine Möglichkeit geschaffen, Trainingsdaten zu erzeugen, solche neuronalen Netze zu trainieren.

Eine andere Art, das Problem des abstrakten Denkens anzugehen, ist die Verwendung der induktiven Logikprogrammierung (ILP). Bei diesem Paradigma des maschinellen Lernens besteht die Aufgabe darin, ein logisches Programm zu konstruieren, um gegebene Daten zu erklären - die natürlich dazu neigen, abstrakt und allgemein zu sein. In diesem Bereich arbeite ich sowohl an der Verbesserung bestehender ILP-Methoden als auch an der Integration von ILP mit Deep Learning.

Contents

Chapter 1

Introduction

The field of Artificial Intelligence has seen great advances with the use of Deep Neural Networks (DNN) [LBH15]. However, the problem of creating a system capable of abstract reasoning remains unsolved [Cho19, CCE+18].

In [Cho19] an abstract reasoning challenge was introduced, which provides an example of a task that current AI systems are incapable of, even though it is trivial for humans. Some examples of such tasks are shown in Figure 1.1. The main difficulty (from the point of view of DNNs) is the lack of a large training set – instead the AI has to infer the pattern from a small number of examples.

Large Language Models (LLM) [RWC+19, BMR+20, ZRG+22, SFA+22] have been shown to perform some reasoning necessary in text generation, and even perform simple algorithms when properly prompted [BMR+20] but still struggle with logical relations between statements. In [RKB+22] the authors show that LLM stuggle to infer whether an indirect answer to a question meant *yes* or *no* (for example, "The sun was scorching." as an answer to "Was that hot?" means *yes*).

These models use an encoder part of the Transformer architecture [VSP+17] to predict the next word in a natural language text. This task of language prediction requires the model to be able to perform a great variety of tasks, which includes at least some abstract reasoning.

One of the most promising current attempts at forcing Deep Neural Networks to perform strict reasoning consists of giving a Language Model a few examples of an algorithm being performed and prompting it to continue [NAGA+21, ZNL+22]. This approach does seem to provide much better results over just asking for an answer, but still seems to fall apart after a dozen steps. When prompted to reason step by step such models are prone to giving incorrect inferences or asserting false claims (often referred to as *hallucinating*).

This proves that while current AI systems are certainly very impressive, there is still work to be done on their ability to perform strict reasoning. This thesis focuses on various methods of providing Deep Neural Networks with the ability to manipulate and construct complex abstractions, which in turn would allow for general reasoning well outside of a training distribution.

One way to force AI to perform abstract reasoning is to directly learn an abstract task – such as Automated Theorem Proving. In ATP, the goal is to construct a formal proof of a mathematical statement. This requires finding a correct sequence of inferences in an exponentially large space of possibilities. This search can be guided by a Deep Neural

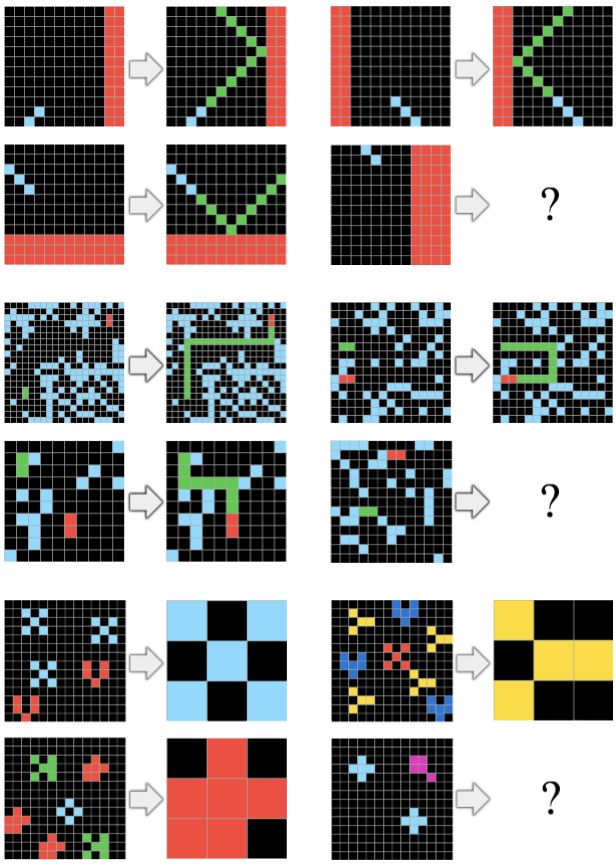


Figure 1.1: Examples of tasks from the Abstraction and Reasoning Challenge [Cho19]

$$\frac{\vdash P \quad \vdash P \to Q}{\vdash Q}$$

Figure 1.2: Example formal inference rule – modus ponens

Network. Such a network would necessarily need to use mathematical abstractions.

To allow guidance of Automated Theorem Provers with Deep Neural Networks, I have worked on creating a neural architecture that would work well with mathematical formulas [Pur20]. I also develop an algorithm that allows for training such networks without a need for human-generated data [PK22], using a general game-playing algorithm [SHS+17].

A different way of tackling the abstract reasoning problem is to use Inductive Logic Programming [CD20, Mug91]. In this machine learning paradigm, the task is to construct a logic program to explain given data – which naturally tends to be abstract and general.

It is worth mentioning that the task of program synthesis is in some way the same as proof synthesis, through Curry-Howard isomorphism [BR67, How69]. It is not however directly transferable, since this isomorphism is between proving a statement and writing a functional program of a given type. In ILP, for one, we construct a logic program, and also have to make it fit some data examples, rather than some type.

In the field of ILP, I am working on improving the existing ILP methods and integrating ILP with Deep Learning. I show how to improve a recent ILP system called "Popper" [CM21a] by allowing it to use of higher-order predicates in its output programs [PCK22b]. I also work on merging ILP methods with DNNs by improving differentiable ILP system δILP [EG18] by using a much larger dimensional search space for gradient descent [PCK22a].

1.1 Formal mathemathics

Mathematical reasoning can be formalized into a set of formally defined rules [RW10]. This allows writing down mathematical proofs that can be mechanically verified to be correct.

Having such proof does not mean the statement is absolutely certain and true, since there still remains a question of whether the rules themselves are correct – a question that can never be answered [Göd92]. Still, formal proofs can be used to verify the correctness of critical programs and circuitry, giving greater confidence in them, as well as finding potential errors.

This work is only concerned with the problem of finding such formal proof, that can be programmatically verified.

1.2 Automated Theorem Proving

Automated Theorem Proving (ATP) is a method of formally proving statements automatically, using a computer program. There are many limitations to this approach: for

$$\dfrac{\dfrac{\dfrac{}{p \wedge (p \to q) \vdash p \wedge (p \to q)}}{p \wedge (p \to q) \vdash p} \quad \dfrac{\dfrac{}{p \wedge (p \to q) \vdash p \wedge (p \to q)}}{p \wedge (p \to q) \vdash p \to q}}{\dfrac{(p \wedge (p \to q)) \vdash q}{\vdash (p \wedge (p \to q)) \to q}}$$

Figure 1.3: Example formal proof in sequent calculus

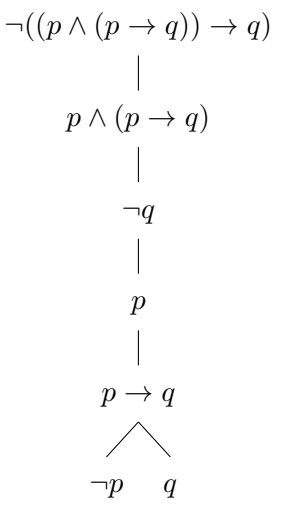


Figure 1.4: Example Tableux proof

example, the problem of deciding whether a statement is provable is undecidable – it is even possible that the statement is true without being provable [Göd92].

It is technically possible to write a program that would find a proof of any provable statement – for example, by searching through all possible proofs in increasing size – but the program would run forever if the statement is unprovable. Moreover, this approach would be highly intractable. In practice, the best results are achieved with heuristic algorithms.

1.2.1 Saturation provers

A well-performing category of Automated Theorem Provers are *Saturation provers*. These work by constructing an ever-expanding set of proven statements – consequences of the input axioms. Usually, the prover attempts to prove *false*, thus proving that all the axioms given are together contradictory. Therefore if we want to prove that some conjecture is a consequence of some set of axioms, we add this conjecture to the set of axioms *negated*. If from this set we can prove a contradiction, we know that if the axioms hold, the conjecture must also hold (since the negated conjecture cannot hold).

Examples of saturation provers are E prover [SCV19] and Vampire [RV02].

1.2.2 Tableaux provers

Tableaux provers construct a branching tree of statements, with branching representing different possible ways of satisfying given statements. For example, a statement $p \vee q$ can be split into branches where p holds in one and q holds in the other.

To complete a proof each branch has to be closed by connecting two contradicting statements, thus proving that all given statements cannot be true together. As before, to prove a conjecture we add it negated as an assumption. An example of such proof is in Figure 1.4. It starts with the negated conjecture at the top ($\neg((p \wedge (p \rightarrow q)) \rightarrow q)$). The consequences of this conjecture are added below, in the same branch because a negated implication being true means that both left side is true and right side is false. Similarly, consequences of a conjunction ($(p \wedge (p \rightarrow q))$) are added. Lastly, consequences of a (non-negated) implication ($p \rightarrow q$) are added in separate branches (because an implication only guarantees one of those must hold). Then, both the branches contain a statement with its negated above, thus finishing the proof.

The important choice when constructing a Tableaux is which statement's consquences to use next. This choice is additionally made more difficult in First-Order, where a statement may need to be used multiple times, with different instantiations.

1.3 Interactive Theorem Proving

Since creating a computer program capable of proving any theorem in general, there have been developed approaches that keep humans in the loop. Such *Interactive Theorem Provers* (ITP) (also referred to as *proof assistants*) are systems in which people construct the proofs, possibly with assistance from ATPs. Such proofs are also immediately verified. Examples of ITPs are Isabelle [NPW02], Coq [BC13] and Mizar [BBG$^+$18].

While this work does not deal directly with ITPs, it does use datasets of human-constructed proofs that were constructed in these systems.

1.4 Guiding Automated Theorem Provers

In some ways, machine learning is a natural fit for theorem proving. For one, we do not really need to care about testing robustness [GR14], as we have a built-in, absolute way of testing performance – are the correct proofs successfully constructed. While it may be useful during research, we do not really need to understand why do our heuristics give the answers they do, as long as we can verify the final proof. There are no consequences for the predictions being inaccurate, other than the failure to construct a proof.

Another reason to turn to machine learning is the fact that the problems in theorem proving are in general undecidable. If we still want to be able to solve such problems automatically, we have to use heuristics. Hand-crafted methods rarely (if ever) reach the performance of a human working on a problem. At the same time, certain machine learning methods achieved and surpassed human-level performance for certain tasks [SHM$^+$16].

Finally, this direction of Artificial Intelligence research promises to create systems capable of abstract reasoning, as mathematical problems (almost by definition) require such capabilities. Therefore whatever solutions work well in this domain, would probably help generally in creating AI systems capable of abstract reasoning.

1.4.1 Neural models for formulas

Currently, the most promising method of machine learning are Deep Neural Networks. Therefore, it is an obvious idea to apply them to the task of guiding a theorem prover. However, mathematical formulae are not a typical input for a neural network, which generally process images and natural language text.

Recursive Neural Networks

Recursive Neural Networks (RNN) use a neural architecture where a neural layer is applied recursively to the input (repeatedly, with the same weights). A simple example is a model which takes as input a sequence of symbols, processing them one by one, using the same network repeatedly (see figure 4.2).

A more complex way of using RNNs, is using the tree structure that mathematical formulae naturally have. In this approach, the network is used to aggregate embeddings of subtrees, recursively, in the end computing an embedding of the whole tree.

Graph Neural Networks

One natural idea for processing mathematical concepts with neural networks is to represent them as graphs. However, unlike image processing, where convolutions and pooling is a well-performing standard, there is no such architecture for Graph Neural Networks (GNN).

A commonly used pattern is *message passing*, where every layer combines information from neighboring nodes to compute the next layer's node embeddings. The method of combining can range from simple summing up embeddings of neighboring nodes to using an attention mechanism [VCC+17].

This method however has several drawbacks. For one, information can only travel one edge per layer, so combining information from far away nodes requires many layers. This problem is made worse by the fact that GNNs tend to perform badly with a large number of layers. Another problem is the fact that such a network can recognize graphs only up to Weisfeiler–Lehman isomorphism test [WL68, XHLJ18], meaning that if the test says the graphs are isomorphic, the networks will process the graphs as if they really were exactly the same — even if they are not.

Autoencoders

Autoencoders [Kra91] are neural networks trained to compute the identity function on some data. Their architecture usually contains some bottleneck, which forces the network to learn patterns present in the data, to be able to reconstruct everything from smaller

bottleneck information. This also means that all information about the input needs to be somehow represented within the bottleneck, which is the property we use in this work.

1.4.2 Training data for theorem proving

To actually make use of a neural network we need to be able to provide it with training data. One obvious approach is to use human-generated data and do imitation learning. However, if we want to be able to solve difficult problems, merely imitating humans might not be enough.

It would be better to learn from the ground truth, that is straight from mathematical results. Some attempts at this have been made in the meantime, like [KUMO18, BSR+19] where an algorithm tries to prove theorems from a given dataset and learns from successful attempts – this however requires a robust dataset of theorems and presents a clear limit of what can be learned. A different attempt [FAA+21] generates a dataset of synthetic theorems and learns from it, thus removing the need for a human-provided dataset, however, this allows for only a shallow exploration of possible theorems.

Mizar40 dataset

Mizar40 dataset [KU15] is extracted from the mathematical library of the Mizar proof system [BBG+18]. The library covers all major domains of mathematics and includes several proofs from theorem proving. As such, we believe that it is representative of the capability of the developed encodings to generalize to mathematical theorem proving. The dataset is structured as follows. Each theorem (goal) is linked to two sets of theorems. One set, the positive examples, are theorems useful in proving the original theorem, and one set, the negative examples, is a set of theorems that were not used in proving the goal. Note that for each theorem its positive and negative example sets are the same size. The negative examples are selected by the nearest neighbor heuristic.

TPTP

Thousands of Problems for Theorem Provers (TPTP) [Sut17] is a library of problems for Automated Theorem Provers. It was collected with the purpose of testing and evaluating ATP systems. It contains problems from multiple domains, from general algebra and graph theory to biology and medicine.

The domains differ in the problems themselves as well as vocabulary that is used to state said problems. For instance, in the set theory problem set one would find predicates such as `member`, `subset`, and `singleton` whereas in the category theory dataset has predicates such as `v1_funct_2`, and `k12_nattra_1`.

1.4.3 Adversarial training

Adversarial training is a machine learning method where one trains both a solver to the task that we actually want to solve and an *adversary* that competes against the actual solver.

Such approaches can be greatly varied. For example in Generate Adversarial Networks [GPAM+20] there are two DNNs trained: a generator, that generates images (the task that we actually want) and a discriminator, that is trained to discriminate between real images and the images generated by the generator. The generator has to learn to deceive the discriminator, which eventually leads to generating images that look good to humans.

In AlphaZero [SHS+18] the algorithm learns to play two-player board games (like Chess or Go) by playing against itself. There the adversary and the solver are one and the same, but playing two different sides of the game.

1.4.4 AlphaZero algorithm

The AlphaZero [SHM+16] algorithm learns by playing the game against itself, then uses the generated games to learn and improve. Then it generates better games and so on. This allows a deep exploration of interesting and valuable areas in the space of possible games by continuously using already learned knowledge to learn more.

The learned part of the algorithm is a Deep Neural Network that estimates the value and policy of any given game state. *Value* is simply the estimated result of the game when in a given state, while *policy* is a vector, describing a probability distribution over available actions. This vector is learned via reinforcement learning, to maximize the reward, defined by the outcome of a game.

1.4.5 Monte-Carlo Tree Search

To facilitate learning in this process the games need to be played a little better than the currently learned the value and policy estimations would allow. Without that, there would be nothing to learn from the playouts.

This is achieved by using Monte-Carlo Tree Search (MCTS). It is an exploration of a tree of game states, checking possible ways the game could go, biased towards states that the the neural network trained thus far estimates as most likely.

The exact formula used in AlphaZero [SSS+17] for choosing the next state to explore is

$$\Big(\log \frac{n_{parent} + c_{base} + 1}{c_{base}} + c_{init}\Big) \frac{\sqrt{n_{parent}}}{n_{child} + 1} \pi_{child} + v_{child}$$

where

- c_{base} and c_{init} are hyperparameters
- n_{parent} and n_{child} are numbers that the parent and the child node were explored already
- π_{child} is the probability given by policy predicting network
- v_{child} is the value of the considered child state as estimated by the MCTS tree explored thus far

BK	E^+	E^-
$\texttt{mom}(a,b).\ \texttt{dad}(e,b).$ $\texttt{mom}(a,c).\ \texttt{dad}(e,c).$ $\texttt{mom}(b,d).\ \texttt{dad}(c,f).$	$\texttt{gp}(a,d).\ \texttt{gp}(e,d).$ $\texttt{gp}(a,f).$	$\texttt{gp}(a,b).\ \texttt{gp}(b,c).$ $\texttt{gp}(c,f).\ \texttt{gp}(d,f).$

Figure 1.5: Example Inductive Logic Programming problem: *grandparent*

$\texttt{gp}(X,Y)\texttt{:-mom}(X,C),\texttt{mom}(C,Y)$	$\texttt{gp}(X,Y)\texttt{:-p}(X,C),\texttt{p}(C,Y)$
$\texttt{gp}(X,Y)\texttt{:-mom}(X,C),\texttt{dad}(C,Y)$	$\texttt{p}(X,Y)\texttt{:-mom}(X,Y)$
$\texttt{gp}(X,Y)\texttt{:-dad}(X,C),\texttt{mom}(C,Y)$	$\texttt{p}(X,Y)\texttt{:-dad}(X,Y)$

Figure 1.6: Example solutions to the problem in figure 1.5

Whenever the algorithm wants to explore another node, it starts from the root and goes down to a child with the highest score (defined by the formula above). After reaching a leaf, a new game state is computed, and a neural network is used to estimate its value and policy (v and π).

This formula means that to start with, the tree is explored proportionally to the predicated policy π. This is also what would happen if all states had the same value (each state would be visited a number of times proportional to policy). However, the formula also takes into account values v, so the exploration is biased towards states with high values (with the balance between value and policy dictated by c parameters).

1.5 Inductive Logic Programming

Inductive Logic Programming (ILP) [CD20] is a machine learning paradigm that explicitly uses abstract reasoning – therefore investigating it and potential uses of it together with Deep Learning is a promising direction towards an AI capable of abstract reasoning. In ILP the goal is to induce a logic program that fits the given training data. While its performance on many AI tasks is worse than that of Deep Learning, it has some advantages. One of which is the ability to generalize from a small number of examples, by constructing general, abstract rules.

A task in ILP is defined by its *background knowledge* (BK), along with positive (E^+) and negative (E^-) examples. An example task is shown in figure 1.5 with possible solutions shown in figure 1.6.

Usually, the ILP systems use constraints of the space of possible logic programs (solutions) to make the search easier. An example of such constraint would be to only consider programs defining a single predicate – which would exclude the solution on the right in figure 1.6. These constraints are referred to as *language bias*.

```
sum(X,Y):-zero(Z),fold(add,X,Z,Y).

sum(X,Y):-empty(X),zero(Y).
sum(X,Y):-tail(X,Z),sum(Z,Q),
          head(X,U),add(Q,U,Y).
```

Figure 1.7: Two programs implementing the same predicate with and without Higher-order predicates.

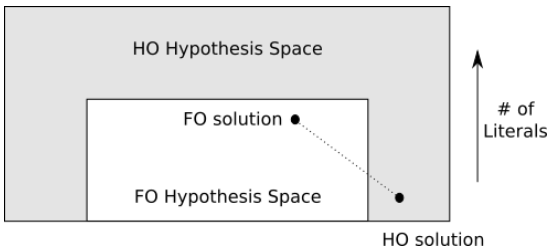


Figure 1.8: Inclusion of HO definitions increases the size of the search space, but can lead to the search space containing a shorter solutions.

1.5.1 Learning From Failures paradigm

A recently developed ILP system is "Popper" [CM21a], which works by iteratively constraining the search space of possible programs.

Whenever a program is generated that does not fit the training examples, it either accepts something it should not, or does not accept something it should. In the first case, other programs that accept everything this program does are also known to not be the solutions, and can therefore be skipped – a constraint is added that forbids generating such programs. Similarly, when the program rejects something it should accept, other programs that accept no more than this program are forbidden.

After adding a constraint a new program is generated, and the loop continues until either a solution is found or the search space is exhausted.

1.5.2 Higher-Order ILP

Allowing ILP systems to use higher-order predicates in their solutions has multiple potential benefits. For one, by allowing the constructed program to use HO predicates, we make solving a lot of problems much easier, even if the search space is increased by introducing new background predicates. This is because, with the use of HO, the target program can be much smaller – making it easier to find (see figures 1.7 and 1.8).

Another potential benefit is that many abstract concepts can be encoded as higher-order predicates and a system capable of using them can potentially benefit from additional layers of abstraction.

Higher-order support has been implemented for two ILP systems: Metagol and HEXMIL [CMM20]. The authors have proven both theoretically and experimentally that using higher-order can reduce the size of the program and the number of examples needed to

learn it. An extension of a more recent ILP system Popper [CM21a] is done in this work in chapter 6.

1.5.3 Differentiable ILP

A method of performing ILP by stochastic gradient descent [SMDH13] was proposed by Evans and Grefenstette [EG18] (called δILP). This approach can potentially allow for merging ILP and Deep Neural Networks together into one system trainable end-to-end.

This approach works by using fuzzy logic operators to compute inferences of a logic program. Instead of simply being *true* or *false*, the values of predicates for given arguments are represented as real numbers between 0 and 1.

Because the system limits itself only to the case with a finite number of atoms, the inference of a logic program can be expressed as a simple logic formula, which can be computed using fuzzy operators. This allows computing a loss value as binary cross-entropy between the expected value (1 for positive examples and 0 for negative) and the output of the system. Loss in turn allows for computing gradients (since fuzzy operators are differentiable) that allow for finding a program through gradient descent.

To use gradient descent the logic program needs to be represented in a continuous parameter space. This is achieved by splitting the program that we want to construct into parts (in the original work the parts were simply separate predicate definitions) and creating a list of all code snippets that could be used for this part. For each part, we then create a parameter representing a distribution of all code snippets. During computation of the learned program, whenever we would have to use a part of the code, we compute values for every code snippet that the part could be. Then, we take a weighted average, using the distribution from a parameter as weights. To allow for recursion we execute this computation multiple times – this number of times is effectively a recursion limit.

Fuzzy truth values computes in this way are differentiable, and allow for computation of gradients on the distribution parameters. These can be used to find which code snippets create a correct program.

1.6 Content

In this dissertation, in chapter 3, I present my work on improving Graph Neural Networks. Then, in chapter 4, I try to use autoencoding to learn useful embeddings on mathematical formulae.

Chapter 5 presents my work on creating a learning algorithm that would learn to prove theorems without human guidance, using adversarial methods.

The later part deals with Inductive Logic Programming, in chapter 6, I show an augmentation of an existing, recently published ILP system that allows it to use higher-order predicates. Later, I deal with an older ILP system, δILP, that learns through gradient descent (the same as neural networks). I show that adding additional invented predicates during learning (increasing the dimensionality of the search space) is beneficial in this ILP method.

Chapter 2

Contributions

The following chapter describes the publications included in this dissertation as chapters. The details about the publication venue and my contributions to each publication are outlined below.

2.1 Improving Expressivity of Graph Neural Networks

Publication Details

> Stanislaw J. Purgal. Improving expressivity of graph neural networks. *2020 International Joint Conference on Neural Networks (IJCNN)*, pages 1–7, 2020

In this chapter, I investigate the limitations (and ways of overcoming them) of Graph Neural Networks – namely the Weisfeiler-Lehman isomorphism limitation [WL68] and far-away interactions. I augment a Transformer-based [VSP+17] Graph Attention Network [VCC+17] by two modifications:

- random initial node embeddings — to facilitate the attention mechanism recognizing different nodes, we add (by concatenation) a random vector to the initial embedding of every node, with different random values every time the embeddings are evaluated
- expanding attention window — we use multi-headed attention [VSP+17], with separate heads for different edge categories. Some of the attention heads only see neighbors (as is standard), but some see exponentially expanding neighborhoods (nodes in distance 2, 4, 8 and so on)

I prove experimentally that these modifications successfully deal with the problems mentioned above as they help classify synthetic datasets built to expose these weaknesses. When testing on a real-world chemical dataset, however, the modifications do not improve the results.

My contribution

I am the sole author of this particular paper: I have proposed the described model, implemented it, performed the experiments, and wrote the article.

2.2 A Study of Continuous Vector Representations for Theorem Proving

Publication Details

Stanisław Purgał, Julian Parsert, and Cezary Kaliszyk. A study of continuous vector representations for theorem proving. *Journal of Logic and Computation*, 02 2021

In this chapter, we investigate the idea of using an autoencoder [Kra91] architecture to train embedding for mathematical formulae. To this end, we train an encoder and a decoder such that the tree shape of a formula including all symbols can be reconstructed from the dense vector representation. We do that by training multiple decoder parts: one that extracts the top symbol of the tree and one that extracts embedding vectors of subtrees. We propose and evaluate two ways of training such autoencoders. One of the approaches is to optimize the difference between a subtree encoding and the output of a decoder. The second one is to optimize the chance of all symbols being correctly predicted after recursively decoding the formula.

The syntactic and semantic logical properties that we aim to preserve include both structural formula properties, the applicability of natural deduction steps, and even more complex operations like unifiability. We propose datasets that can be used to train these syntactic and semantic properties. We evaluate the viability of the developed encoding across the proposed datasets as well as for the practical theorem proving problem of premise selection in the Mizar corpus.

My contribution

I have proposed and implemented two ways of training a formula autoencoder. I have then performed experiments testing the reconstruction ability as well as the performance of a simple classifier neural network trained to use our pre-trained encoders. I have also written the corresponding parts of the article.

2.3 Adversarial Learning to Reason in an Arbitrary Logic

Publication Details

Stanisław J. Purgał and Cezary Kaliszyk. Adversarial learning to reason in an arbitrary logic. *The International FLAIRS Conference Proceedings*, 35, May 2022

We propose Monte-Carlo simulations guided by reinforcement learning that can work in an arbitrarily specified logic, without any human knowledge or set of problems. Since the algorithm does not need any training dataset, it can learn to work with any logical foundation, even when there is no body of proofs or even conjectures available.

The approach uses the AlphaZero [SHS+17] algorithm to learn a strategy for a proposed theorem proving game. This game has one player (*the adversary*) constructing a provable

theorem and the second one (*the prover*) trying to prove it.

We train a neural network in the way AlphaZero does, without relying on any human data (other than the definition of the logic system). After some training, we test the capability of the prover using a human-generated set of theorems.

For a state estimation neural network, we use a variant of the Graph Neural Network described in chapter 3.

We practically demonstrate the feasibility of the approach in multiple logical systems. The approach is stronger than training on randomly generated data but weaker than the approaches trained on tailored axiom and conjecture sets. It however allows us to apply machine learning to automated theorem proving for many logics, where no such attempts have been tried to date, such as intuitionistic logic or linear logic.

My contribution

I have proposed and implemented the system discussed in this paper, as well as performed the experiments. I have written the parts of the article describing how the system works and the experiments performed (sections 5.4 to 5.8).

2.4 Learning Higher-Order Logic Programs from Failures

Publication Details

> Stanisław J. Purgał, David M. Cerna, and Cezary Kaliszyk. Learning higher-order logic programs from failures. In Lud De Raedt, editor, *Proceedings of the Thirty-First International Joint Conference on Artificial Intelligence, IJCAI-22*, pages 2726–2733. International Joint Conferences on Artificial Intelligence Organization, 7 2022. Main Track

We augment the recently introduced *Learning From Failures* paradigm by allowing it to use higher-order definitions.

The existing system *Popper* [CM21a] works in a loop, by generating a program, inferring constraints (in the program space) by checking how the program fails, then generating a new program, taking new (and old) constraints into account.

We extend this system by allowing it to use higher-order definitions, like *fold* or *map*. We do this by making sure that whenever a HO predicate is used, a predicate needs to be defined as its argument.

We prove that as long as the HO predicate is monotone with respect to *subsumption* and *entailment* ($p_1 \leq_\theta p_2 \Rightarrow H(p_1) \leq_\theta H(p_2)$) the constraint system from *Popper* remains sound. Most of examples of HO predicate (*fold* and *map*) are monotone.

Experimental results show that our extension significantly improves learning performance without the burdensome human guidance required by existing systems. Our theoretical framework captures a class of higher-order definitions preserving the soundness of existing subsumption-based pruning methods.

My contribution

I have proposed a way of synthesizing a higher-order logic program using the *Learning From Failures* paradigm and described the limitations of this method (the requirement of higher-order predicate to be monotonic). I have then implemented the method, using code published by the authors of [CM21a]. I have also described my contributions in the paper.

2.5 Differentiable Inductive Logic Programming in High-Dimensional Space (contibution beyond the PhD)

The following work is not yet published but is also something I have worked on during my PhD.

There is a problem with the δILP system [EG18], that it easily gets stuck in local minima. We have hypothesized that increasing the dimensionality of the search space would solve this problem. This increase was achieved by increasing the number of *invented predicates* – additional predicates that can be used in the constructed program.

Because this increase would result in infeasibly large memory usage, we also modified the system to split the constructed program into smaller parts. This approach was considered by the original authors [EG18] in an appendix to their work. They did not use it because it would result in even bigger problems with local minima – a problem that we aim to overcome anyway.

Partially for this change, but also to improve efficiency we have developed new implementation of δILP, using PyTorch [PGM+19] library, that works much faster than the existing version – both because of the improved implementation and smaller parts.

Chapter 3

Improving Expressivity of Graph Neural Networks

3.1 Abstract

We propose a Graph Neural Network with greater expressive power than commonly used GNNs — not constrained to only differentiate between graphs that Weisfeiler–Lehman test recognizes to be non-isomorphic. We use a graph attention network with expanding attention window that aggregates information from nodes exponentially far away. We also use partially random initial embeddings, allowing differentiation between nodes that would otherwise look the same. This could cause problems with a traditional dropout mechanism, therefore we use a "head dropout", randomly ignoring some attention heads rather than some dimensions of the embedding.

3.2 Introduction

Recently there has been a great interest in neural network architectures capable of processing graphs [YJK+19, DHS+19, ZXC+18, YBY+19, XWZ+19]. They are applied for tasks of molecule properties prediction [HKKS01], premise selection in theorem proving [WTWD17], RNA sequence classification [RMBL19] etc.

Most Graph Neural Networks (GNNs) can recognize graphs only up to Weisfeiler–Lehman isomorphism test (WL-test) [WL68, XHLJ18], meaning that if the test says the graphs are isomorphic, the networks will process the graphs as if they were exactly the same — even if they are not.

In our work, we seek to overcome two types of failure of the WL-test. First is when the difference between graphs is only noticeable when considering long connections (eg. as in fig. 3.1). Another failure that we correct is when we need to notice whether two indirect connections lead to one and the same node or to two similar nodes (as in fig. 3.2).

The first failure is addressed in our proposed model by aggregating nodes with an exponentially expanding window. This way we allow the network to notice a connection of exponential length. This operation could be seen as an attempt to imitate operations done in usual convolutions, such as pooling done in computer vision, which also aggregates information from exponentially far away, although in a more structured way. Another

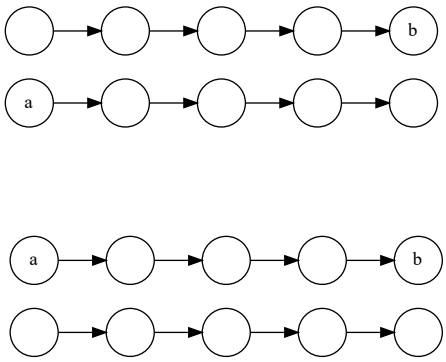


Figure 3.1: Graphs consisting of two paths

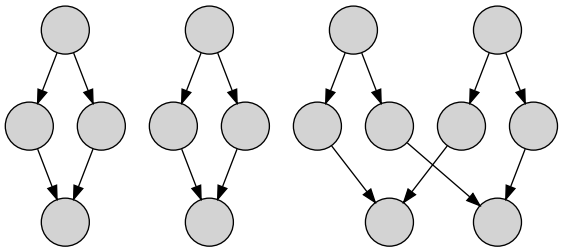


Figure 3.2: "Diamond" graphs that common GNNs cannot differentiate

operation we can be said to imitate is an expanding dilated convolution used in WaveNet [vdODZ+16], which again aggregates information from far away. Both those approaches use the intrinsic structure of the data to aggregate more information layer by layer rather than trying to process larger and larger sets. Unfortunately, in general, there is no such structure in graphs.

The second problem of the WL-test is solved by introducing a random identifier for every node present in the graph. This preserves invariance under node permutation while allowing the network to differentiate between nodes even if they all look the same — thus allowing graph attention to be used even when no labels are present.

3.3 Preliminaries

We assume the reader to be familiar with the self-attention mechanism [VSP+17] and its use in graph attention networks [VCC+17].

Graphs considered in this work are directed, with labeled nodes and edges. We allow

all labels in a graph to be equal. Where we consider symmetric graphs, we model it with directed graphs where for every edge there exists a symmetric edge in the other direction. We do not consider multi-edges, though technically our model allows for edges with multiple labels.

When presenting formulas for calculations done in our model we mark parts with learnable parameters with subscript ϕ. We use $||$ to mark concatenation and $\odot$ to mark point-wise multiplication (or Hadamard product).

3.4 Proposed model

Our proposed model modifies standard graph attention [VCC+17] in two ways:

- random initial node embeddings — to facilitate the attention mechanism recognizing different nodes, we add (by concatenation) a random vector to the initial embedding of every node, with different random values every time the embeddings are evaluated.

- expanding attention window — we use multi-headed attention [VSP+17], with separate heads for different edge categories. Some attention heads only see neighbors (as is standard), but some see exponentially expanding neighborhoods (nodes in distance 2, 4 and so on).

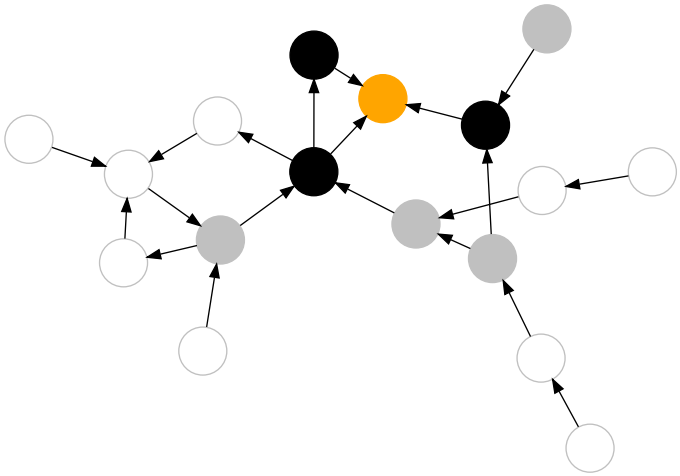
3.4.1 Partially random initial node embeddings

In our model (expGNN), the initial embedding of a node is composed of two concatenated components of the same length. One is a learnable embedding of a node label, the other a *random node identifier*, a random vector composed of 1s and 0s (each possible with probability $\frac{1}{2}$).

$$\mathbf{n}^0_{i\phi} = \textsc{Embed}_\phi(\textsc{Label}(n_i)) \;||\; \textsc{RandomSequence}(\{0 : \frac{1}{2}, 1 : \frac{1}{2}\})$$

This identifier is different every time an embedding is being calculated but stays the same within one graph instance. This means that it is possible to differentiate between nodes, even if their label and neighborhoods are the same.

3.4.2 Expanding attention window



To facilitate the propagation of information within a graph (faster than one edge per one layer), we propose an *expanding attention window*. In each layer, this window expands exponentially, aggregating information from nodes further away. So, in layer n we aggregate nodes that are within distance 2^n.

3.4.3 Multiple attention filters

Since it is not clear that this expanding window would be helpful for every task, we use different windows for different attention heads, with some aggregating only neighbors, some using this expanding window, and some aggregating from all nodes in the graph. Since we want the information to spread both ways, not only in the direction of edges, we also use different heads where edges go in the opposite direction.

All attention head types used in our model are:

- Neighbouring nodes (different edge types separately)
- Reversed neighboring nodes (all edge types together)
- Expanding window (all edge types together)
- Reversed expanding window (all edge types together)
- All nodes in the graph

When working with an adjacency matrix, expanding the window can be done quite efficiently, by calculating a new adjacency matrix:

$$A_{n+1} = \min(1, A_n \cdot A_n + A_n)$$

Of course, when working with more optimized graph representations for sparse graphs, this operation is very costly, as it makes the graph much denser.

3.4.4 Single layer architecture

For a single layer in our model we use residual connection [HZRS16], similar to that used in Transformer [VSP+17], but also utilizing layer normalization [BKH16].

$$\begin{aligned}\mathbf{n}_{i\phi}^{n+1} = \textsc{ReLU}(&\mathbf{n}_{i\phi}^{n} + \mathrm{FNN}_{\phi}(\mathbf{n}_{i\phi}^{n} || \\ &\textsc{FilteredMultiHead}_{\phi}^{n}(\mathbf{n}_{i\phi}^{n}, \mathbf{n}_{*\phi}^{n}, \mathbf{n}_{*\phi}^{n})))\end{aligned}$$

$$\begin{aligned}\mathrm{FNN}_{\phi}(x) = &\textsc{NormalizedLayer}_{\phi}(\\ &\textsc{ReLU}(\textsc{NormalizedLayer}_{\phi}(x)))\end{aligned}$$

Multi-headed dot-product attention works as in [VSP+17], the only difference being using different masks for different heads.

$$\begin{aligned}\textsc{Attention}_{\phi}(Q,K,V,M) &= \alpha V W_{\phi}^{V} \\ \alpha &= \textsc{maskedSoftmax}(M,\beta) \\ \beta &= \frac{(QW_{\phi}^{Q})(KW_{\phi}^{K})^{T}}{\sqrt{d_k}} \\ \textsc{maskedSoftmax}(M,x) &= \frac{\exp x \odot M}{\sum \exp x \odot M}\end{aligned}$$

$$\begin{aligned}\textsc{FilteredMultiHead}_{\phi}^{n}(Q,K,V) = &\textsc{concat}(\\ &\textsc{Attention}_{\phi}(Q,K,V,M_1) \\ &\vdots \\ &\textsc{Attention}_{\phi}(Q,K,V,M_h))\end{aligned}$$

Normalized layer is defined in [BKH16] as:

$$\begin{aligned}\textsc{NormalizedLayer}_{\phi}(x) &= \frac{g_{\phi}}{\sigma} \odot (a-\mu) + b_{\phi} \\ a &= xA_{\phi} \\ \mu &= \frac{1}{H}\sum_{i=1}^{H} a_i \\ \sigma &= \sqrt{\frac{1}{H}\sum_{i=1}^{H}(a_i-\mu)^2}\end{aligned}$$

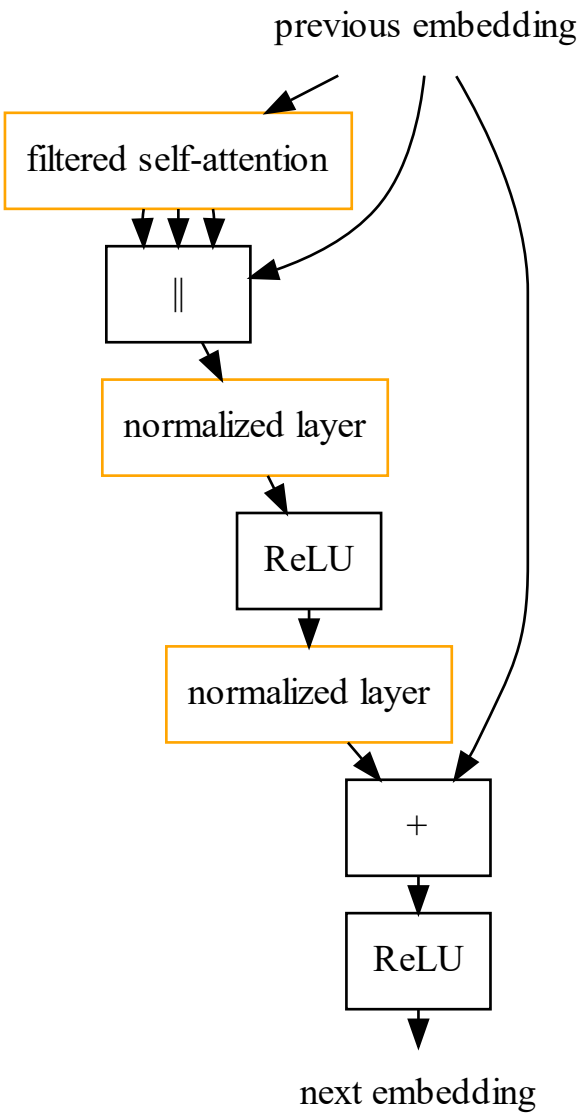


Figure 3.3: Single layer architecture

3.4.5 Final aggregation for graph classification

The node embeddings resulting from a few layers described above (in our experiment 3) are aggregated from all nodes in the graph using simple *maximum*. The resulting graph embedding is fed to a two-layer feed-forward network.

3.4.6 Head dropout

The standard dropout [SHK+14] mechanism may conflict with the random initial embeddings. The network is supposed to rely on the random distribution of vector representations of the nodes in the graph. Using dropout changes this distribution, making it different during training and during evaluation. This could (and a did few times during the experiments) lead to a situation where loss goes down while the accuracy remains poor.

To counteract this problem, and to force learning of different useful properties, we use a "head dropout". Instead of removing some parts of vectors, we randomly ignore certain attention heads. During training, each type of attention window (immediate neighbors, expanding, reversed etc.) is ignored with some probability (in our experiments 0.1).

3.5 Experiments

We test the ability of our model to recognize properties that theoretically require overcoming the limitation of the WL-test. To do that, we generate artificial datasets, with graph labels determined by the tested property. To better validate generalizing ability, we use more than one evaluation set, with a few different methods of generating random graphs (but using the same property for labels).

For training datasets, we use uniform random graphs, where every edge exists with the same probability. This probability is chosen to be such that about half of the generated graphs have the property being tested.

The size of training datasets is 10^6 (one million), and the sizes of random testing datasets are all 10^4 (ten thousand). The synthetic datasets used are available online[1] in a format compatible with [KKM+16].

3.5.1 Presence of a cycle in a symmetric graph

We generate symmetric graphs with 32 nodes and classify them by checking whether there is a cycle in the graph.

Evaluation sets include:

- more random graphs from the same distribution as the training set
- uniform random graphs with 64 nodes (with lower edge-existence probability) and with 16 nodes (with higher edge-existence probability)

[1] `http://cl-informatik.uibk.ac.at/cek/ijcnn2020/`

- random trees
- random trees with one additional edge (creating a cycle)
- line graphs of length between 3 and 64
- cycles of length between 3 and 64

Random trees are generated by adding nodes one by one, attaching each one to a random already existing node. Half of such generated trees also receive one additional edge between a random pair of not connected nodes.

3.5.2 Presence of a clique 4

For training, again, we use random uniform graphs with 16 nodes. Evaluation sets include also bigger (and sparser) graphs than those used in training.

In each set, the class on a graph depends on the presence of a clique 4 (a subset of 4 nodes where each node is connected to every other node).

3.5.3 Categorizing circulant skip links

We test our network on the dataset the most difficult dataset used in [MSRR19, DSSV19]. This dataset has 10 categories, with only 1 graph each. Each graph is a $\mathcal{G}_{\text{skip}}(41, R)$ with R being one of $\{2, 3, 4, 5, 6, 9, 11, 12, 13, 16\}$. A graph $\mathcal{G}_{\text{skip}}(N, R)$ contains N nodes $\{1, ..., N\}$, such that a pair of nodes (a, b) is connected if (and only if) $|a - b| \equiv 1$ or $R \pmod N$ (see fig. 3.4 for an example).

In their tests [MSRR19, DSSV19] use 15 randomly permuted instances of each graph (for a total of 150 graphs in the dataset). Since our network is invariant under permutations, that would be pointless here, and we only use 10 graphs. For evaluation, however, since our model is non-deterministic, we do use 150 graphs to get a better evaluation of our accuracy.

Since no generalizing beyond the training set is necessary for this test, we do not use dropout here.

3.5.4 Presence of a path from one highlighted node to the other

As earlier, the training set consists of uniformly random graphs, now with two nodes being given special labels (a and b). The class of a graph depends on the existence of a path from a to b. In the training set, all graphs have 32 nodes.

As a special testing case, we use graphs consisting of two paths. The highlighted nodes can be either on the ends of one path or on two different paths (shown is figure 3.1). Those graphs are very similar and hard for commonly used GNNs to differentiate between. We use paths of lengths from 2 to 32.

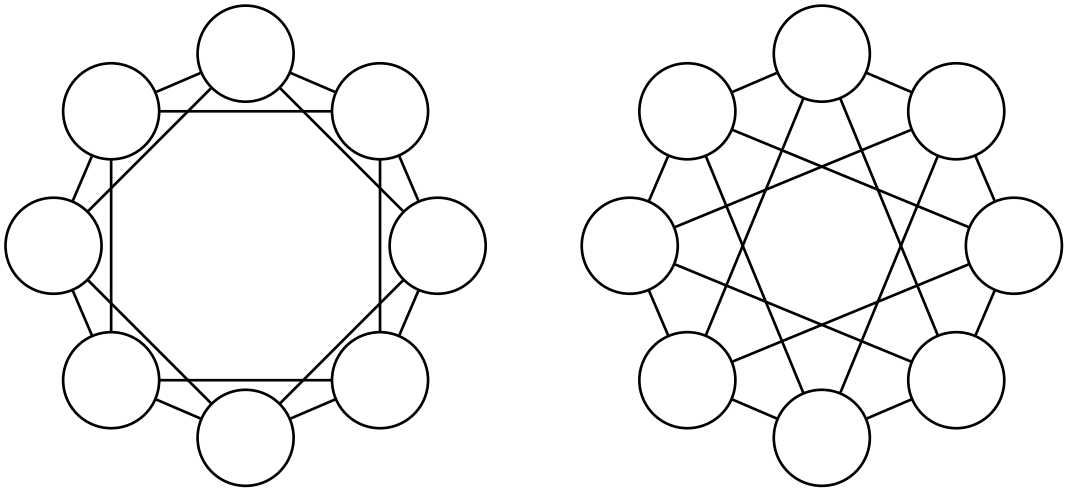


Figure 3.4: Circulant skip links — $\mathcal{G}_{\text{skip}}(8, 2)$ and $\mathcal{G}_{\text{skip}}(8, 3)$

Table 3.1: Presence of a clique 4 results

model	accuracy			
	training	size 16	size 32	size 64
GFN	0.8076	0.8142	0.5068	0.5092
GCN	0.9005	0.9088	0.5118	0.4887
GraphStar	0.9983	**0.9772**	0.5331	0.5092
expGNN	0.9129	0.9108	0.7349	**0.5662**
expanding window only	0.5016	0.4924	0.4955	0.4908
random init only	0.9293	0.9279	**0.7401**	0.5427
basic graph attention	0.5016	0.4924	0.4955	0.4908

	accuracy						
	training	uniform 32	uniform 64	uniform 16	trees 64	trees 32	lines + cycles
GFN	0.9947	0.9961	0.9099	0.8737	0.7659	0.8141	0.0887
GCN	0.9995	**0.9996**	0.8914	0.8768	0.5649	0.8755	0.2419
GraphStar	0.9983	0.9729	**0.9994**	**1.0000**	0.8167	0.9005	0.1452
expGNN	0.9990	0.9993	0.9819	0.9999	**0.8227**	**0.9499**	0.5726
expanding window only	0.5294	0.5275	0.4424	0.5106	0.4951	0.4982	0.5000
random init only	0.9823	0.9836	0.9230	0.9972	0.7950	0.9226	**0.6129**
basic graph attention	0.5294	0.5275	0.4424	0.5106	0.4951	0.4982	0.5000

Table 3.2: Presence of a cycle results

Table 3.3: Circulant skip links results

model	accuracy			
	mean	std	max	min
RP-GIN [MSRR19]	0.376	0.129	0.533	0.100
16-CLIP [DSSV19]	0.908	0.068	0.987	0.760
Ring-GNN [CVCB19]	N/A	0.157	0.800	0.100
expGNN	**0.978**	0.015	**0.993**	**0.947**
expanding window only	0.100	0.000	0.100	0.100
random init only	0.687	0.030	0.740	0.647
basic graph attention	0.100	0.000	0.100	0.100

Table 3.4: Presence of node of degree 7 results

model	accuracy		
	training	size 16	size 32
GFN	1.0000	**1.0000**	**1.0000**
GCN	1.0000	**1.0000**	0.8300
GraphStar	1.0000	**1.0000**	**1.0000**
expGNN	0.9520	0.9534	0.5903
expanding window only	0.5863	0.5906	0.5385
random init only	0.9862	0.9873	0.6402
basic graph attention	0.5863	0.5906	0.5385

Table 3.5: Presence of a path results

model	accuracy				
	training	size 16	size 32	size 64	paths
GFN	0.8358	0.8453	0.8276	0.6775	0.5483
GCN	0.9706	0.7057	0.9696	0.6810	0.5161
GraphStar	0.9979	1.0000	0.9975	0.9925	0.5967
expGNN	1.0000	**1.0000**	**1.0000**	**1.0000**	0.8371
expanding window only	1.0000	**1.0000**	**1.0000**	0.9999	**0.8629**
random init only	0.9903	0.9984	0.9889	0.9853	0.5645
basic graph attention	0.9881	0.9977	0.9866	0.9859	0.5806

Table 3.6: Chemical datasets results

model	accuracy		
	SN12C	MOLT-4	Yeast
GFN	0.9639	0.9374	**0.8899**
GCN	0.9592	0.9336	0.8871
GraphStar	0.9640	**0.9394**	0.8884
expGNN	0.9633	0.9335	0.8870
expanding window only	**0.9656**	0.9365	0.8875
random init only	0.9626	0.9347	0.8865
basic graph attention	0.9648	0.9365	0.8870

3.5.5 Presence of a node with 7 neighbors

In this dataset, we simply generate uniform graphs and check whether there is a node with a degree of 7 or greater. For training, we use graphs of size 16, for testing we use also bigger graphs of size 32.

3.5.6 Chemical datasets

We also test our model on a few chemical datasets from [KKM+16]. These were published on [Yan], collected from the PubChem website[2]. Each dataset belongs to a certain type of cancer screen with the outcome active or inactive.

3.5.7 Tested models

For comparison with our model we use several recently published graph neural architectures: Graph Feature Network[CBS19], Graph Convolutional Network (using the implementation from the same work [CBS19]) and Graph Star Net [LHYG19].

The exception is the experiment with a circulant skip list, where those networks mathematically can't differentiate between graphs. There we compare with results reported in other papers that also used this dataset [MSRR19, DSSV19, CVCB19]. Since [CVCB19] does not report the mean of their results, we leave it as "N/A".

We also test variants of our model with only one of the two modifications, as well as without both (making it a Graph Attention Network [VCC+17]).

3.5.8 Hyperparameters

In our model, we use 3 layers of graph message passing. In every layer, each node is encoded in 128 dimensions. In dot-product attention, the queries and keys have 32 dimensions. Each type of attention head is used thrice. During training, each type has

[2] `https://pubchem.ncbi.nlm.nih.gov/`

a 0.1 chance of being ignored. For optimization we use Adam optimizer [KB14] with default $\beta_1 = 0.9$, $\beta_2 = 0.999$, $\epsilon =$ 1e-7 and learning rate 1e-3.

3.6 Results and discussion

3.6.1 Presence of a clique and a cycle

These two observed graph properties are on one hand simple, on the other according to [XHLJ18] cannot really be expressed by usual GNNs. Somewhat surprisingly, results in tables 3.1 and 3.2 show that GNNs still learn to recognize them with high accuracy given a graph of the same size as those in the training set. However, changing the size of the graph and the density of edges greatly lowers the accuracy, revealing that the learned property is not actually what we wanted.

Our proposed model seems to be able to generalize the property to graphs of different sizes much better.

3.6.2 Categorizing circulant skip links

The results in table 3.3 show that our model achieves better accuracy than reported in [MSRR19, DSSV19, CVCB19].

We see that categorizing long skip links is impossible in 3 layers when not using the expanding attention window. With it, however, even 3 layers are enough.

We note that [CVCB19] also reports 100% accuracy with Ring-GNN-SVN, a variant of Ring-GNN that is given top eigenvalues of adjacency matrices, allowing for trivial classification.

3.6.3 Presence of a node with degree 7 and presence of a path

The last two graph properties are things that can be trivially learned by some GNNs. For most used models, the most basic property is the degree of a node (trivially extracted, or in GFN [CBS19] just given as part of the initial embedding). In our model, learning to extract the degree of a node is possible, but much harder (and, because it depends on random initial embeddings, remains not 100% accurate). Instead, the basic property is detecting a connection.

3.6.4 Chemical datasets

Experiments on chemical datasets (results shown in table 3.6) show that even though our model has higher theoretical expressive power, it does not improve accuracy on chemical benchmarks.

3.7 Related work

This work seeks to improve Graph Neural Networks. The core idea of GNNs [SGT+08] is to generate new node embeddings by aggregating embeddings of neighboring nodes.

Initially, a recurrent network would process the nodes until their embeddings converged to some value. Currently, most networks use a constant number of layers that aggregate nodes (as do we). GraphSAGE [HYL17] experiments with aggregating embeddings using simple functions like mean and maximum.

Following spectral graph theory, Kipf et al. [KW16] propose a Graph Convolution operator. It can be thought of as a sum aggregation, but with embeddings scaled by an inverse of a square root of a node degree ($\frac{1}{\sqrt{d_n}}$ or $d_n^{-\frac{1}{2}}$), both before and after aggregation. In [CBS19] some features are added to the initial node embeddings.

Graph Attention Networks, a type of GNNs that we build on, were introduced in [VCC+17]. In this network attention mechanism, [BCB14] is used to aggregate the embeddings. Attention extracts information from a set of vectors (representations of things — in our case nodes) by first estimating the importance of every element of the set and then calculating the weighted average (with weights depending on importance). The importance calculation can be done using a smaller feedforward neural network that given the context and the element estimates the importance of the element in the context, or (as in [VSP+17]) by calculating a dot-product of some projection of context representation with a projection of the element.

The attention mechanism allows for aggregating information of a *set*, rather than a *sequence* (that is, ignore ordering of elements), which fits exactly what we need in graph processing.

All of the above-mentioned networks allow information to travel only one edge per network layer. To allow far information propagation a Graph Star Net was proposed [LHYG19], where a global state (a few "star" nodes) is updated in every layer. This allows information to propagate globally and to neighbors, but not anything in-between. Thus, this network still suffers from the constraint of the WL-test. Our model allows also far-but-not-global propagation, it is however much more computationally costly for large graphs.

3.7.1 On Graph Attention without node labels

The output of an attention mechanism with a multiset of exactly the same elements as the input will always be equal to that one element. This is because the output of attention is essentially a weighted average, and if all elements are the same, the output will be the same regardless of what (and how many) the weights are.

Because of this, in a simple graph attention network, when all the nodes have the same embedding (eg. when no node labels are provided), they will remain the same after however many layers. The network can only differentiate between having any neighbors and having none.

The GraphStar network [LHYG19] gets around this problem by using attention across both neighbors and a few global stars. In this way, after one layer the embedding depends on the degree of a node (effectively nodes are given labels based on their degree).

Our proposed model uses random node identifiers, making a situation where all nodes have the same embeddings extremely unlikely (effectively impossible).

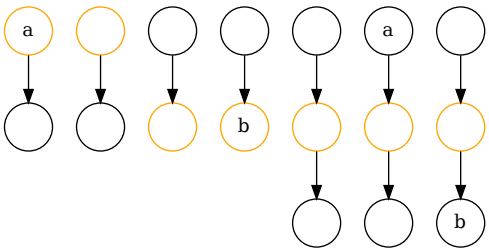


Figure 3.5: All immediate neighborhoods of a two paths graph (the same regardless of whether a and b are connected)

3.7.2 Perception limits of GNNs

Xu et al. [XHLJ18] describe the expressive power of message-passing GNNs as equivalent to the Weisfeiler-Lehman isomorphism test [WL68]. This means that a node embedding can depend only on the node's subtree structure of a certain depth (the depth being equal to the number of layers). The graph classification then depends on the multiset of subtree structures present in the graph. A network capable of distinguishing between all multisets of subtree structures (of certain depth) is referred to in [XHLJ18] as a *maximally powerful GNN*. Yet even such networks cannot differentiate between graphs that WL-test deems isomorphic.

The work of [MSRR19] seeks to overcome this problem by using using a permutation-sensitive aggregator and summing over all permutations. To make this computationally feasible, they propose k-ary Relational Pooling.

A different approach is proposed in [DSSV19] where they use one-hot encoded coloring of nodes added in a way that allows for differentiating between nodes.

A mechanism similar to our expanding window was described in [CLB17], where multiple powers of adjacency matrix were used during aggregation (in their experiments they were A^2 and A^4). Later in [CVCB19] they use a learnable mechanism to calculate consecutive powers of adjacency matrix, that can in particular learn to express the property very similar to our exponentially expanding window (what the model in [CVCB19] can express is $\min(A^{2^n}, 1)$, a window containing all nodes that can be reached in *exactly* 2^n steps). In this variant, the "adjacency matrices" don't necessarily contain only 1s and 0s.

Our work seeks to expand the limit of the WL-test in two places: for one, by utilizing expanding attention window we effectively increase the depth of subtree structures exponentially. Since we also use direct neighborhood in some attention heads we theoretically don't lose any expressive power. Another way our model is more expressive is its ability to recognize the connection to one and the same node from a connection to two identical nodes. We achieve this by using random initial embeddings.

We should point out that because of the use of randomness we lose the property of isomorphic graphs always having the same embedding — instead we have isomorphic

graphs having the same *distribution* of embeddings.

3.8 Conclusion

We present a Graph Neural Network with more expressive power than any model we have seen described. We show its ability to differentiate between graphs that other networks cannot. What our models seem to excel at is classifying synthetic datasets of graphs WL-test fails to recognize as non-isomorphic and generalization to previously unseen graph sizes (and edge densities).

Future work includes translating this improvement to accuracy on chemical datasets.

Acknowledgement

This research was supported by the ERC starting grant no. 714034 SMART.

Chapter 4

A Study of Continuous Vector Representations for Theorem Proving

4.1 Abstract

Applying machine learning to mathematical terms and formulas requires a suitable representation of formulas that is adequate for AI methods. In this paper, we develop an encoding that allows for logical properties to be preserved and is additionally reversible. This means that the tree shape of a formula including all symbols can be reconstructed from the dense vector representation. We do that by training two decoders: one that extracts the top symbol of the tree and one that extracts embedding vectors of subtrees. The syntactic and semantic logical properties that we aim to preserve include both structural formula properties, the applicability of natural deduction steps, and even more complex operations like unifiability. We propose datasets that can be used to train these syntactic and semantic properties. We evaluate the viability of the developed encoding across the proposed datasets as well as for the practical theorem proving the problem of premise selection in the Mizar corpus.

4.2 Introduction

The last two decades saw an emergence of computer systems applied to logic and reasoning. Two kinds of such computer systems are interactive proof assistant systems [HUW14] and automated theorem proving systems [RV01]. Both have for a long time employed human-developed heuristics and AI methods, and more recently also machine learning components.

Proof assistants are mostly used to transform correct human proofs written in standard mathematics to formal computer-understandable proofs. This allows for the verification of proofs with the highest level of scrutiny, as well as automatic extraction of additional information from the proofs. Interactive theorem provers (ITPs) were initially not intended to be used in standard mathematics, however, subsequent algorithmic developments and modern-day computers allow for a formal approach to major mathematical proofs [Hal08]. Such developments include the proof of Kepler's conjecture [HAB+17] and the four colour theorem [Gon08]. ITPs are also used to formally reason about computer systems, e.g.

have been used to develop a formally verified operating system kernel [KAE+10] and a verified C compiler [Ler09]. The use of ITPs is still more involved and requires much more effort than what is required for traditional mathematical proofs. Recently, it has been shown that machine learning techniques combined with automated reasoning allow for the development of proofs in ITPs that is more akin to what we are used to in traditional mathematics [BKPU16].

Automated reasoning has been a field of research since the sixties. Most Automated Theorem Proving systems (ATPs) work in less powerful logics than ITPs. They are most powerful in propositional logic (SAT solvers), but also are very strong in classical first-order logic. This is mostly due to a good understanding of the underlying calculus and its variants (e.g. the superposition calculus for equality [BGLS92]), powerful low-level programming techniques, and the integration of bespoke heuristics and strategies, many of which took years of hand-crafting [SCV19, Vor14].

In the last decade, machine learning techniques became more commonly used in tools for specifying logical foundations and for reasoning. Today, the most powerful proof automation in major interactive theorem proving systems filter the available knowledge [KvLT+12] using machine learning components (Sledgehammer [BGK+16], CoqHammer [CK18]). Similarly, machine-learned knowledge selection techniques have been included in ATPs [KSUV15]. More recently, techniques that actually use machine learning to guide every step of an automated theorem prover have been considered [UVŠ11, LISK17] with quite spectacular success for some provers and domains: A leanCoP strategy found completely by reinforcement learning is 40% more powerful than the best human developed strategy [KUMO18], and a machine-learned E prover strategy can again prove more than 60% more problems than the best heuristically found one [CJSU19]. All these new results rely on sophisticated characterizations and encoding of mathematics that are also suitable for learning methods.

The way humans think and reason about mathematical formulas is very different from the way computer programs do. Humans familiarize themselves with the concepts being used, i.e. the context of a statement. This may include auxiliary lemmas, alternative representations, or definitions. In some cases, observations are easier to make depending on the representation used [GAA+13]. Experienced mathematicians may have seen or proven similar theorems, which can be described as intuition. On the other hand, computer systems derive facts by manipulating syntax according to inference rules. Even when coupled with machine learning that tries to predict useful statements or useful proof steps the reasoning engine has very little understanding of a statement as characterized by an encoding. We believe this to be one of the main reasons why humans are capable of deriving more involved theorems than modern ATPs, with very few exceptions [KVV13].

In this paper, we develop a computer representation of mathematical objects (i.e. formulas, theorem statements, proof states), that aims to be more similar to the human understanding of formulas than the existing representations. Of course, human understanding cannot be directly measured or compared to a computer program, so we focus on an approximation of human understanding as discussed in the previous paragraph. In particular, we mean that we want to perform both symbolic operations and "intuitive steps" on the representation. By symbolic operations, we mean basic logical

inference steps, such as modus ponens, and more complex logical operations, such as unification. When it comes to the more intuitive steps, we would like the representation to allow direct application of machine learning to proof guidance or even conjecturing. A number of encodings of mathematical objects as vectors have been implicitly created as part of deep learning approaches applied to particular problems in theorem proving [ACE+16, WTWD17, OKU19]. However, none of them have the required properties, in particular, the recreation of the original statement from the vector is mostly impossible.

It may be important to already note, that it is impossible to perfectly preserve all the properties of mathematical formulas in finite-length vectors of floating-point values. Indeed, there are finitely many such vectors and there are infinitely many formulas. It is nonetheless very interesting to develop encodings that will preserve as many properties of as many formulas as possible, as this will be useful for many practical automated reasoning and symbolic computation problems.

Contribution We propose methods for supervised and unsupervised learning of an encoding of mathematical objects. By encoding (or embedding) we mean a mapping of formulas to a continuous vector space. We consider two approaches: an explicit one, where the embedding is trained to preserve a number of properties useful in theorem proving and an implicit one, where an autoencoder of mathematical expressions is trained. For this several training datasets pertaining to individual logical properties are proposed. We also test our embedding on a known automated theorem proving problem, namely the problem of premise selection. We do so using the Mizar40 dataset [KU15]. The detailed contributions are as follows:

- We propose various properties that an embedding of first-order logic can preserve: formula well-formedness, subformula property, natural deduction inferences, alpha-equivalence, unifiability, etc. and propose datasets for training and testing these properties.
- We discuss several approaches to obtaining a continuous vector representation of logical formulas. In the first approach, representations are learned using logical properties (explicit approach), and the second approach is based on autoencoders (implicit approach).
- We evaluate the two approaches for the trained properties themselves and for a practical theorem proving problem, namely premise selection on the Mizar40 dataset.

The paper extends our work presented at GCAI 2020 [PAK20], which discussed the explicit approach to training an embedding that preserves properties. The new material in this version comprises an autoencoding of first-order logic (this includes the training of properties related to decoding formulas), new neural network models considered (WaveNet model and Transformer model), and a more thorough evaluation. In particular, apart from the evaluation of the embeddings on our datasets, we also considered a practical theorem proving problem, namely premise selection on a standard dataset.

Contents The rest of this paper is structured as follows. In Section 4.3 we introduce the logical and machine learning preliminaries. In Section 4.4 we discuss related work. In Section 4.5 we present two methods to develop a reversible embedding: the explicit approach where properties are trained together with the embedding and the implicit approach where autoencoding is used instead. In Section 4.6 we develop a logical properties dataset and present the Mizar40 dataset. Section 4.7 contains an experimental evaluation of our approach. Finally Section 4.8 concludes and gives an outlook on future work.

4.3 Preliminaries

4.3.1 Logical Preliminaries

In this paper, we will focus on first-order logic (FOL). We only give a brief overview, for a more detailed exposition see Huth and Ryan [HR00].

An abstract Backus-Naur Form (BNF) for FOL formulas is presented below. The two main concepts are terms (4.1) and formulas (4.2). A formula can either be an Atom (which has terms as arguments), two formulas connected with a logical connective, or a quantified variable or negation with a formula. Logical connectives are the usual connectives negation, conjunction, disjunction, implication and equivalence. In addition, formulas can be universally or existentially quantified.

$$\text{term} := \text{var} \mid \text{const} \mid f(\text{term}, \ldots, \text{term}) \tag{4.1}$$

$$\begin{aligned} \text{formula} := {} & \text{Atom}(\text{term}, \ldots, \text{term}) \\ & \mid \neg\text{formula} \mid \text{formula} \wedge \text{forumla} \\ & \mid \text{formula} \vee \text{formula} \\ & \mid \text{formula} \rightarrow \text{formula} \mid \text{formula} \leftrightarrow \text{formula} \\ & \mid \exists \text{ var. formula} \mid \forall \text{ var. formula} \end{aligned} \tag{4.2}$$

For simplicity we omitted rules for bracketing. However, the "standard" bracketing rules apply. Hence, a formula is well-formed if it can be produced by (4.2) together with the mentioned bracketing rules. The implementation is based on the syntax of the FOL format used in the "Thousands of Problems for Theorem Provers" (TPTP) library [Sut17][1]. This library is very diverse as it contains data from various domains including set theory, algebra, natural language processing and biology all expressed in the same logical language. Furthermore, its problems are used for the annual CASC competition for automated theorem provers. Our data sets are extracted from and presented in TPTP's format for first-order logic formulas and terms. An example for a TPTP format formula is `![D]: ![F]: (disjoint(D,F) <=> ~intersect(D,F))` which corresponds to the formula $\forall d.\ \forall f.\ \text{disjoint}(d, f) \iff \neg\,\text{intersect}(d, f)$. As part of the data extraction, we developed a parser for TPTP formulas where we took some liberties.

[1]The full BNF is available at: `http://www.tptp.org/TPTP/SyntaxBNF.html`

For example, we allow for occurrences of free variables, something the TPTP format would not allow.

To represent formulas we use labeled, rooted trees. So every node in our trees has some *label* attached to it, and every tree has a special *root* node. We refer to the label of the root as the *top symbol*.

4.3.2 Neural Networks

Neural networks are a widely used machine learning tool for approximating complicated functions. In this work, we experiment with several neural architectures for processing sequences.

Convolutional Neural Networks Convolutional neural networks (Figure 4.1 on page 37) are widely used in computer vision [KSH17] where they usually perform two-dimensional convolutions. However, in our case, the input of the network are string representations of formulas, which is a one-dimensional object. Therefore, we only need one-dimensional convolutions.

In this kind of network, convolutional layers are usually used together with spatial pooling, which reduces the size of the object by aggregating several neighbouring cells (pixels or characters) into one. This is illustrated in Figure 4.1 on page 37.

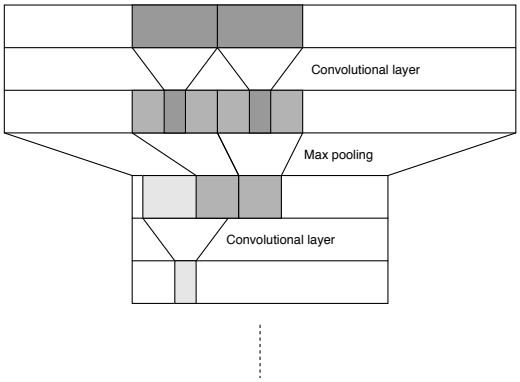


Figure 4.1: Convolutional network

Long-Short Term Memory Long-Short Term Memory networks [HS97] are recurrent neural networks – networks that process a sequence by updating a hidden state with every input token. In an LSTM [HS97] network, the next hidden state is computed using a forget gate, which in effect makes it easier for the network to preserve information in the hidden state. LSTMs are able to learn order dependence, thanks to the ability to retain information long-term, while at the same time passing short-term information between cells. A *bidirectional* network [SP97] processes sequences to directions and combines the final state with the final output.

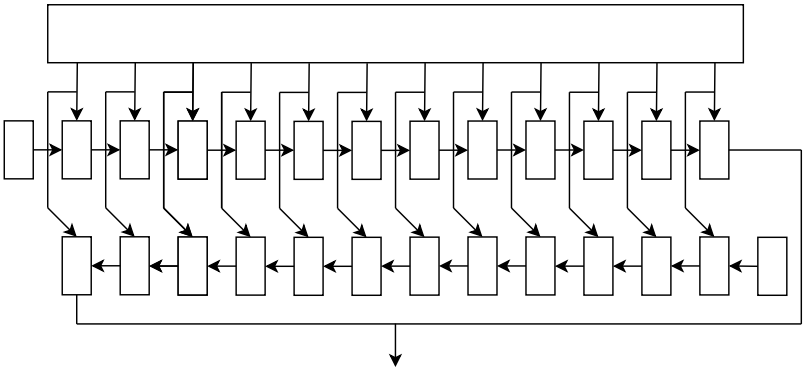


Figure 4.2: Bidirectional LSTM network

WaveNet WaveNet [vdODZ+16] is also a network based on convolutions. However, it uses an exponentially increasing dilation. That means that the convolution layer does not gather information from cells in the immediate neighborhood, but from cells increasingly further away in the sequence. Figure 4.3 on page 38 illustrates how the dilation increases the deeper in the network we are. This allows information to interact across large (exponentially large) distances in the sequence (i.e. formula). This kind of network performed well in audio-processing [vdODZ+16], but also in proof search experiments [LISK17].

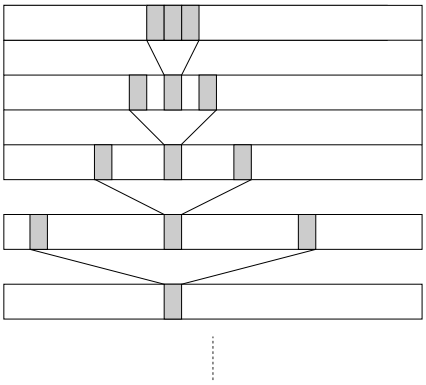


Figure 4.3: WaveNet network

Transformer Transformer networks have been successfully applied to natural language processing [VSP+17]. These networks consist of two parts, an encoder, and a decoder. As we are only interested in encoding we use the encoder architecture of a Transformer network [VSP+17]. This architecture uses the attention mechanism to allow the exchange of information between every token in the sequence. An attention mechanism first computes *attention weights* for each pair of interacting objects, then uses a weighted average of their embeddings to compute the next layer. In Transformer, the weights are

computed as dot-product of "key" and "query" representations for every token. This mechanism is illustrated in Figure 4.4 on page 39.

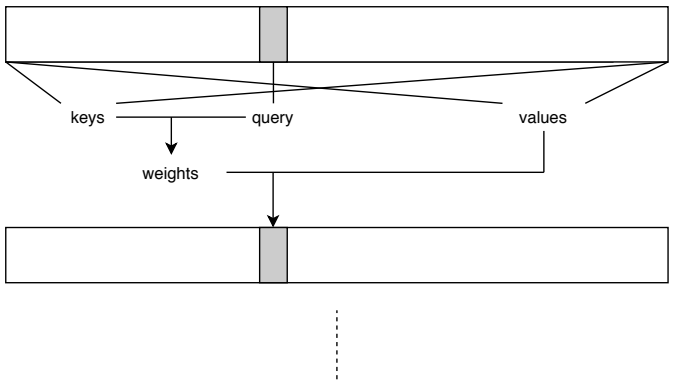


Figure 4.4: Transformer encoder network

Autoencoders [Kra91] are neural networks trained to express identity function on some data. Their architecture usually contains some *bottleneck*, which forces the network to learn patterns present in the data, to be able to reconstruct everything from smaller bottleneck information. This also means that all information about the input needs to be somehow represented within the bottleneck, which is the property we use in this work.

4.4 Related work

The earliest application of machine learning to theorem provers started in the late eighties. Here we discuss only the deep-learning-based approaches that appeared in recent years. As neural networks started being used for symbolic reasoning, specific embeddings have been created for particular tasks. Alemi et al. [ACE+16] have first shown that a neural embedding combined with CNNs and LSTMs can perform better than manually defined features for premise selection. In a setup that also included the WaveNet model, it was shown that formulas that arise in the automated theorem prover E as part of its given clause algorithm can be classified effectively, leading to proofs found more efficiently [LISK17].

Today, most neural networks used for mathematical formulas are variants of Graph Neural Network [HYL17] – a kind of neural network that repeatedly passes messages between neighboring nodes of a graph. This kind of network is applied to the problem of premise selection by Wang et al. [WTWD17]. Later work of Paliwal et al. [PLR+20] experimented with several ways of representing a formula as a graph and also consider higher-order properties.

A most extreme approach to graph neural networks for formulas was considered in [OKU19], where a single hypergraph is constructed of the entire dataset containing all theorems and premises. In this approach, the symbol names are forgotten, instead, all

references to symbols are connected within the graph. This allows constructing the graph and formulate message passing in a way that makes the output of network invariant under reordering and renaming, as well as symmetric under negation. A different improvement was recently proposed by Rawson and Reger [RR19], where the order of function and predicate arguments is uniquely determined by asymmetric links in the graph embedding.

The work of [CAC+19] also uses graph neural networks with message passing, but after applying this kind of operation they aggregate all information using a Tree LSTM network [TSM15]. This allows for representing variables in formulae with single nodes connected to all their occurrences, while also utilizing the tree structure of a formula. A direct comparison with works of this kind is not possible, since in our approach we explicitly require the possibility of decoding the vector back into formulas, and the other approaches do not have this capability.

Early approaches trying to apply machine learning to mathematical formulas have focused on manually defining feature spaces. In certain domains manually designed feature spaces prevail until today. Recently Nagashima [Nag19] proposed a domain-specific language for defining features of proof goals (higher-order formulas) in the interactive theorem prover Isabelle/HOL and defined more than 60 computationally heavy but useful features manually. The ML4PG framework [KHG12] defines dozens of easy to extract features for the interactive theorem prover Coq. A comparison of the different approaches to manually defining features in first-order logic together with features that rely on important logical properties (such as anti-unification) was done by the last author [KUV15]. Continuous representations have also been proposed for simpler domains, e.g. for propositional logic and for polynomials by Allamanis et al. [ACKS17].

We are not aware of any work attempting to auto-encode logical formulas. Some efforts were however done to reconstruct a formula tree. Gauthier [Gau20] trained a tree network to construct a new tree, by choosing one symbol at a time, in a manner similar to sequence-to-sequence models. Here, the network was given the input tree, and the partially constructed output tree and tasked with predicting the next output symbol in a way similar to Tree2Tree models [CAR18]. Neural networks have also been used for translation from informal to formal mathematics, where the output of the neural network is a logical formula. Supervised and unsupervised translation with Seq2Seq models and transformer models was considered by Wang et al. [WBKU20, WKU18], however there the language considered as input was natural language. As such it cannot be directly compared to our current work that autoencodes formulas. Autoencoder-based approaches have also been considered for programming language code, in particular, the closest to the current work was proposed by Dumančić et al. [DGMB19] where Prolog code is autoencoded and operations on the resulting embedding are compared to other constraint solving approaches.

In natural language processing, pre-training on unsupervised data has achieved great results in many tasks [MSC+13, DCLT19]. Multiple groups are working on transferring this general idea to informal mathematical texts, mostly by extending it to mathematical formulas in the ArXiv [YM18]. This is, however, done by treating the mathematical formulas as plain text and without taking into account any specificity of logic.

4.5 Approach

As previously mentioned, our main objective is an encoding of logical formulas. In particular, we are interested in networks that take the string representation of a formula as an input and return a continuous vector representation thereof. This representation should preserve properties and information that is important for problems in theorem proving. We considered two approaches, an implicit and an explicit approach. In the explicit approach, we defined a set number of logical properties (c.f. Section 4.7.2) and related classification problems and trained an encoding network with the loss of these classifiers. The implicit approach is based on autoencoders where we train a network that given a formula encodes it and then decodes it back to the same formula. In theory, this means that the encoding (i.e. continuous vector representation) preserves enough information to reconstruct the original formula. In particular, this means that the tree structure of a formula is learned from its string representation. We will now explain the two approaches in detail starting with the explicit one.

4.5.1 Explicit Approach

The general setup for this approach is depicted in Figure 4.5 on page 42. The green box in Figure 4.5 on page 42 represents an encoding network for which we consider different models which we discuss later in this section. This network produces an encoding $\text{enc}(\phi)$ of a formula ϕ. This continuous vector representation is then fed into classifiers that recognise logical properties (c.f. Section 4.6.1). The total loss $\mathcal{L}$ is calculated by taking the sum of the losses $\mathcal{L}_P$ of each classifier of the properties $p \in \mathcal{P}$ discussed before. $\mathcal{L}$ is then propagated back into the classifiers and the encoding network. This setup is end-to-end trainable and ensures, that the resulting embedding preserves the properties discussed in Section 4.6.1. We train the network on this setup and evaluate the whole training setup (encoding network and classifiers) on unseen data in Section 4.7. However, it is important to note that we are only interested in the encoding network. Hence, we can extract the encoding network (c.f. Figure 4.5 on page 42) and discard the classifiers after training and evaluation. A drawback of this explicit method is that we are working under the assumption that the logical properties that we select are *sufficient* for the tasks that the encodings are intended for in the end. That is, the encodings may only preserve properties that are helpful in classifying the trained properties but not further properties that the network is not trained with. Hence, if the encodings are used for tasks that are not related to the logical properties that the classifiers are trained with, the encodings may be of no use.

Classifiers The classifiers' purpose is to train the encoding network. This is implemented by jointly training the encoding networks and classifiers. There are two philosophies that can go into designing these classifiers. The first is to make the classifiers as simple as possible, i.e. a single fully connected layer. This means that in reality, the classifier can merely select a subspace of the encoding. This forces the encoding networks to encode properties in a "high-level" fashion. This is advantageous if one wants to train simpler

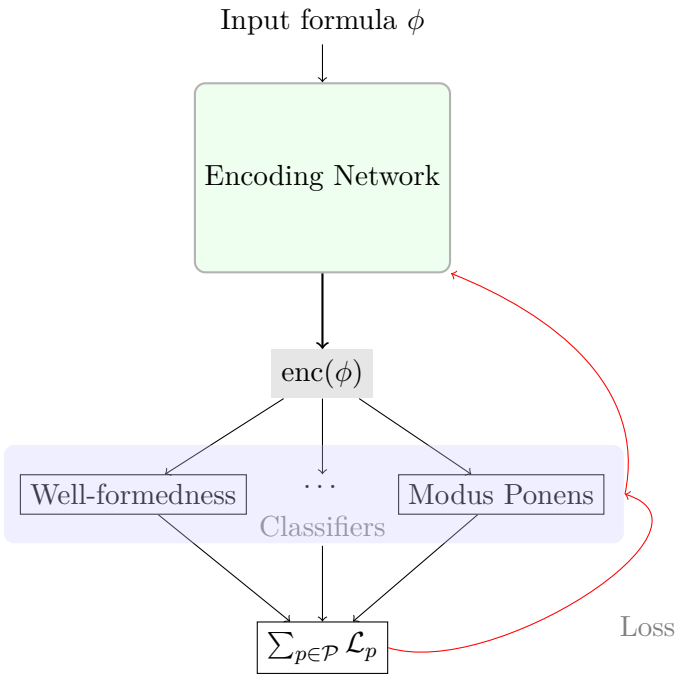


Figure 4.5: The property training framework. The bottom area contains the classifiers that get one or more continuous representations of formulas enc(ϕ) as input. If the classifier takes two formulas as input (i.e. alpha-equivalence), we gather enc(ϕ_1) and enc(ϕ_2) separately and forward the pair (enc(ϕ_1), enc(ϕ_2)) to the classifier. The encoding networks are described subsequently (cf. Figure 4.6 on page 43).

machine learning models with the encodings. On the other hand, when using multiple layers in the classifiers more complex relationships can be recognised by the classifiers and the encoding networks can encode more complex features without having to keep them "high-level". In this scenario, however, if the problems for the classifiers are too easy it could happen that only the classifier layers are trained and the encoding network layers remain "untouched" i.e. do not change the char-level encoding significantly. We chose a middle ground by using two fully connected layers, although we believe that o e could investigate further solutions to this problem (e.g. adding weights to loss).

Encoding Models We considered 20 different encoding models. However, they can be grouped into ten CNN based models and ten LSTM based models. We varied different settings of the models such as embedding dimension, output dimensions as well as adding an additional fully connected layer. The layouts of the two model types are roughly depicted in Figure 4.6 on page 43. The exact dimensions and sizes of the models are discussed in Section 4.7.

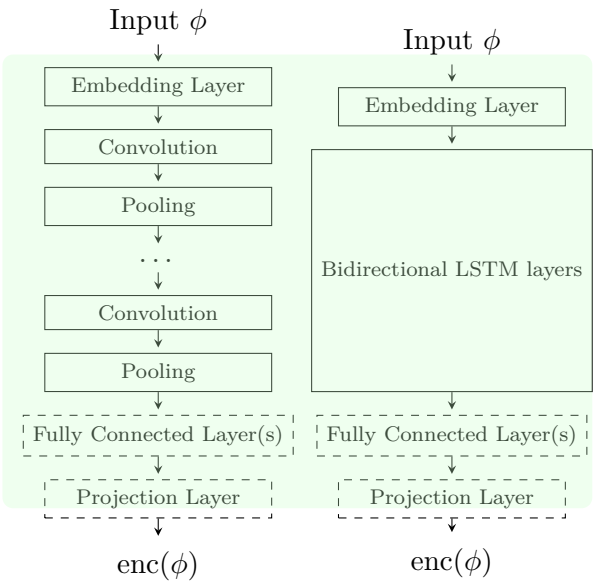


Figure 4.6: The encoding models we considered with the layers that the input passes through. The left diagram depicts CNN-based models, while the right one depicts LSTM-based models. The dashed boxes describe layers that are optional for these model types.

CNN based models The models based on CNNs are depicted on the left in Figure 4.6 on page 43. The first layer is a variable size embedding layer, the size of which can be changed. Once the formulas have been embedded, we pass them through a set of convolution and (max) pooling layers. In our current model, we have 9 convolution and pooling layers with increasing filter sizes and ReLUs as activation functions. The output of the final pooling layer comprises the encoding of the input formula. In the second model, we append an additional set of fully connected layers after the convolution and pooling layers. However, these do not reduce the dimensionality of the vector representation. For that, we introduce a third type of models, which we call embedding models. In embedding models, the last layer is a projection layer which we tested with output dimensions 32 and 64. Note that between the last pooling layer and the projection layer one can optionally add fully connected layers like in the previous model. In Section 4.7 we evaluate these models.

LSTM based models The LSTM based models are depicted on the right side in Figure 4.6 on page 43. Much like in previous models, the first layer is an embedding layer. The output of which gets fed into bidirectional LSTM layers. The output of these layers serves as the encoding of our input formulas. As with the CNN based models, we also considered models where an additional set of fully connected or projection layers is added.

4.5.2 Implicit Approach

As previously mentioned the implicit approach does not work with specific logical properties. We use autoencoders to encode formulas and subsequently retrieve the original formula from the encoding. As such the encoding has to contain enough information about the original formula to reconstruct it from the encoding. Therefore, this method eliminates one of the major drawbacks of the previous approach where the encodings are dependent on the selected logical properties. Figure 4.7 on page 44 depicts a high-level overview of this setup.

We want to train the encoder to generate such continuous vector encodings that can be decoded. For this, we want the possibility to extract top symbol of a formula, as well as the encodings of all its subformulas. These two qualities would indeed enforce the encoding having the complete information about the entire tree-structure of a formula.

To achieve that, we train a top symbol classifier and subtree extractors together with the encoder. The top symbol classifier is a single layer network that given the encoding of a tree classifies it by its top symbol. The subtree extractors are single linear transformations that output an encoding of the i-th subtree. Both encoders and decoders are trained together end-to-end using unlabelled data. As with the explicit approach, we are not interested in decoder networks, and only use them to force the encoder to extract all information from the input. The data (formulas) is provided in a string form but we require the ability to parse this data into trees.

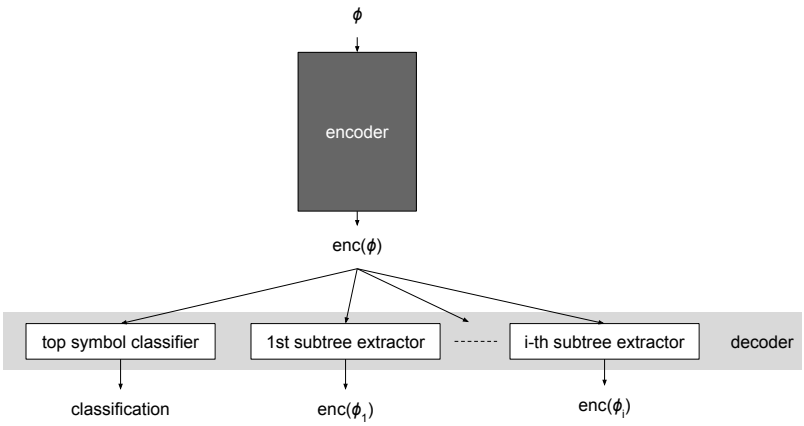


Figure 4.7: Tree autoencoder mechanism

Difference training

Our first approach is to train the top symbol classifier using cross-entropy loss and subtree extractor on mean square error loss using a dataset of all input trees and all their subtrees (Figure 4.8 on page 45).

The first loss is forcing the embedding to contain information about the top symbol, and the second is about the subtrees. In the second loss, we force the result of extracting

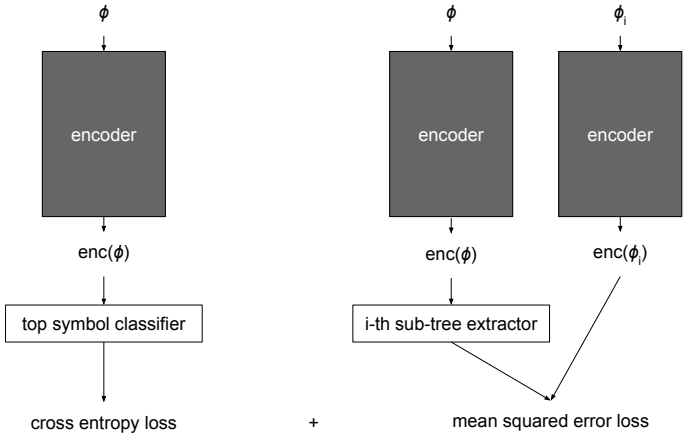


Figure 4.8: Difference training mechanism

a subtree to be equal to the embedding of a subtree itself. Because of this, we need an encoding of the subtree by itself, and for this, we need the input string of a subtree. In formulae datasets, this is generally easy to achieve.

This method of training can be viewed as training on two datasets simultaneously. One dataset consists of formulas with their top symbol, and the other consists of formulas with their i-th subformula and the index i. The first dataset makes sure that the embedding of a formula contains information about its top symbol, and the second one makes sure that the embedding contains information about the embedding of all its subformulas. Together, those requirements force the embedding to contain information about the entire formula, in a form that is easily extracted with linear transformations.

Theoretically minimizing this loss enforces the ability to reconstruct the tree, however, given a practical limit on the size of the encoding, reconstruction fails above a certain tree depth. We do need to restrict the size of the encoding to one that will be useful for practical theorem proving tasks, like premise selection, etc. With such reasonable limits, we will later in the paper see that we can recover formulas of depth up to about 5, which is a very significant part of practical proof libraries.

Recursive training

In this method we only use the cross-entropy loss on top symbol classification. We compute encodings of subtree recursively (using subtree extractor transformations) and classify their top symbols as well (and so on recursively). All classification losses from a batch are summed together into one total loss that is used for back-propagation.

This is similar to tree recursive neural networks [GK96], like Tree LSTM [TSM15] except pushing information in the other direction (from root to leaves) – we reconstruct the tree from embedding and get a loss in every node.

In this approach, gradient descent can learn to recognize top symbols of subtrees even deep down the input tree. It is however much harder to properly parallelize this

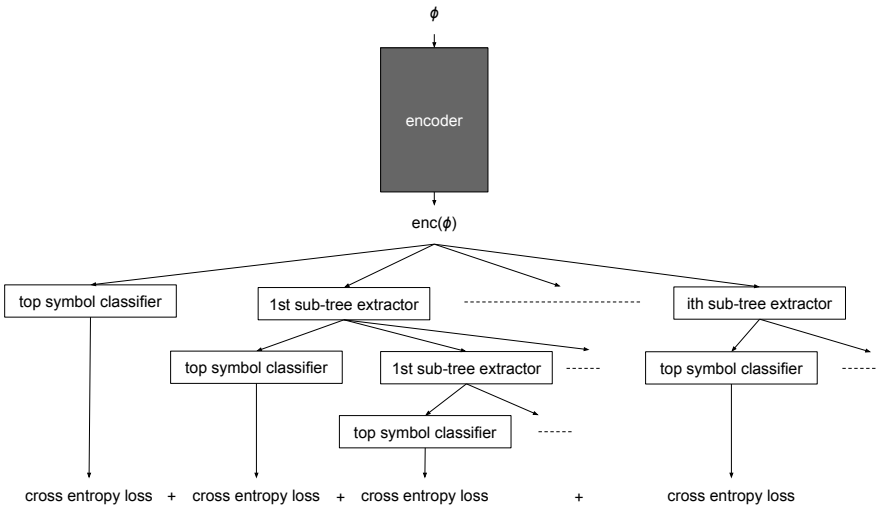


Figure 4.9: Recursive training mechanism

computation, making it much less efficient.

Encoders

As described above the encoding network is independent of the training setup. That is for both, difference and recursive training different encoding models can be used. This is similar to the explicit approach where we also consider different encoding networks. Here, in addition to the already considered *CNN*s and *LSTM*s, we will also consider *WaveNet* and *Transformer* models (introduced in Section 4.3).

All these models receive as input a text string representation of a formula (a character level learned embedding). As output, they all provide a high-dimensional vector representation of a formula.

4.6 Datasets

We will consider two datasets for our training and for the experiments. The first one is a dataset used to train logical properties, that we believe a formula embeddings should preserve. The dataset is extracted from TPTP. TPTP is a database of problems stated in first-order logic. It contains first-order problems from graph theory, category theory, and set theory among other fields. These datasets differ in the problems themselves as well as vocabulary that is used to state said problems. For instance, in the set theory problem set one would find predicates such as `member`, `subset`, and `singleton` whereas in the category theory dataset has predicates such as `v1_funct_2`, and `k12_nattra_1`. The second dataset is the Mizar40 dataset [KU15], a known premise selection dataset. The neural network training part of the dataset consists of pairs of theorems and premises

together with their statements, as well as the information if the premise was useful in the proof or not. Half are positive examples and half are negatives.

4.6.1 Logical properties dataset

We introduce some properties of formulas that we will consider in subsequent sections and describe how the data was extracted.

Well-formedness: As mentioned above it is important that the encoding networks preserve the information of a formula being well-formed. The data set was created by taking TPTP formulas as positive examples and permutations of the formulas as negative examples. We generate permutations by randomly iteratively swapping two characters and checking if the formula is well-formed, if it is not, we use it as a negative example. This ensures that the difference between well-formed formulas and non well-formed formulas is not too big.

subformula: Intuitively, the subformula relation maps formulas to a set of formulas that comprise the original formula. Formally, the subformula relation is defined as follows:

$$\mathrm{sub}(\phi) = \begin{cases} \{\phi\} & \text{if } \phi \text{ is Atom} \\ \mathrm{sub}(\psi) \cup \{\phi\} & \text{if } \phi \text{ is } \neg\psi \\ \mathrm{sub}(\psi_1) \cup \mathrm{sub}(\psi_2) \cup \{\phi\} & \text{if } \phi \text{ is } \psi_1 \wedge \psi_2 \\ \mathrm{sub}(\psi_1) \cup \mathrm{sub}(\psi_2) \cup \{\phi\} & \text{if } \phi \text{ is } \psi_1 \vee \psi_2 \\ \mathrm{sub}(\psi_1) \cup \mathrm{sub}(\psi_2) \cup \{\phi\} & \text{if } \phi \text{ is } \psi_1 \to \psi_2 \\ \mathrm{sub}(\psi_1) \cup \mathrm{sub}(\psi_2) \cup \{\phi\} & \text{if } \phi \text{ is } \psi_1 \leftrightarrow \psi_2 \\ \mathrm{sub}(\psi) \cup \{\phi\} & \text{if } \phi \text{ is } \forall x.\ \psi \\ \mathrm{sub}(\psi) \cup \{\phi\} & \text{if } \phi \text{ is } \exists x.\ \psi \end{cases}$$

Notice, how we never recursively step into the terms. As the name suggests we only recurse over the logical connectives and quantifiers. Hence, $g(x)$ is not a subformula of $\neg f(g(x), c)$ whereas $f(g(x), c)$ is (since "$\neg$" is a logical connective of formulas). Importantly, the subformula property preserves the tree structure of a formula. Hence, formulas with similar sets of subformulas are related by this property. Therefore, we believe that recognizing this property is important for obtaining a proper embedding of formulas. In the presented dataset the original formulas ϕ are taken from the TPTP dataset. Unfortunately, finding negative examples is not as straightforward, since each formula has infinitely many formulas that are not subformulas. In our dataset, we only provide the files as described above (positive examples). To create negative examples during training, we randomly search for formulas that are not a subformula. Since we want to have balanced training data we search for as many negative examples as positive ones.

Modus Ponens: One of the most natural logical inference rules is called *modus ponens*. The modus ponens (MP) allows the discharging of implications as shown in the inference

rule (4.3). In other words, the consequent (right-hand side of implication) can be proven to be true if the antecedent (left-hand side of implication) can be proven.

$$\frac{\overline{P} \quad \overline{P \to Q}}{Q} \tag{4.3}$$

Using this basic inference rule we associate two formulas ϕ and ψ with each other if ϕ can be derived from ψ in few inference steps with modus ponens and conjunction elimination without unification and matching. It turns out that despite its simplicity, modus ponens makes for a sound and complete proof calculus for the (undecidable) fragment of first-order logic known as Horn Formulas [BGG97].

Example 4.1. We can associate the two formulas $\phi := \forall x.\ ((P(x) \to Q(x)) \wedge P(x))$ and $\psi := \forall x.\ Q(x)$ with each other, since ψ can be proven from ϕ using the modus ponens inference rule (and some others).

Providing data for this property required more creativity. We had two approaches: Option one involves generating data directly from the TPTP dataset, while the other option comprised synthesizing data ourselves with random strings. In the data set, we provide both alternatives are used. First, we search for all formulas in the TPTP set that contained an implication and added the antecedent using a conjunction. We paired this formula with the formula containing only the consequent. We tried to introduce heterogeneity to this data by swapping around conjuncts and even adding other conjuncts in-between. Secondly, we synthesize data using randomly generated predicate symbols.

Alpha-Equivalence: Two formulas or terms are alpha equivalent if they are equal modulo variable renaming. For example, the formulas $\forall x\ y.\ P(x) \wedge Q(x, y)$ and $\forall z\ y.\ P(z) \wedge Q(z, y)$ are alpha equivalent. Alpha equivalence is an important property for two reasons. First, it implicitly conveys the notion of variables and their binding. Second, one often works on alpha equivalence classes of formulas, and hence, alpha equivalent formulas need to be associated with each other.

Term vs Formula: We generally want to be able to distinguish between formulas and terms. This is a fairly simple property, especially since it can essentially be read off the BNFs 4.1 and 4.2. However, it is still important to distinguish these two concepts, and a practical embedding should be able to do so.

Unifiability: Unifiability plays an important role in many areas of automated reasoning such as resolution or narrowing [BN98]. Unifiability is a property that only concerns terms. Formally, two terms are unifiable if there exists a substitution σ such that $s \cdot \sigma \approx t \cdot \sigma$. Informally, a substitution is a mapping from variables to terms and the application of a substitution is simply the replacing of variables by the corresponding terms. Formally one needs to be careful that other variables do not become bound by substitutions. Example 4.2 showcases these concepts in more detail.

Example 4.2. Substitution and Unifiability: The terms $t = f(g(x), y))$ and $s = f(z, h(0))$ are unifiable, since we can apply the substitution: $\{z \mapsto g(x),\ y \mapsto h(0)\}$ such that $t \cdot \sigma = f(g(x), h(0)) = f(g(x), h(0)) = s \cdot \sigma$.

Syntactic unification, which is the type of unification described above is quite simple and can be realized with a small set of inference rules. Note that we only consider the relatively simple syntactic unification problem. Interestingly, adding additional information such as associativity or commutativity can make unification an extremely complex problem [BN98]. Putting unification into a higher-order setting makes it even undecidable [Hue02]. Both of these problems could be considered in future work.

4.6.2 Mizar40 dataset

Mizar40 dataset [KU15] is extracted from the mathematical library of the Mizar proof system [BBG+18]. The library covers all major domains of mathematics and includes a number of proofs from theorem proving. As such, we believe that it is representative of the capability of the developed encodings to generalize to theorem proving. The dataset is structured as follows. Each theorem (goal) is linked to two sets of theorems. One set, the positive examples, are theorems useful in proving the original theorem, and one set, the negative examples, is a set of theorems that were not used in proving the goal. Note that for each theorem its positive and negative example set are the same size. The negative examples are selected by a nearest neighbor heuristic[2]. Using this data we generate pairs (consisting of a theorem and a premise) and assign them a class based on whether the premise was useful in proving the theorem.

4.7 Experiments

Since the explicit approach does not allow for decoding formulas, we separately evaluate the two approaches. We first discuss the evaluation of the explicit approach. We first discuss the performance of the different encoding models with respect to the properties, they were trained with as well as separate evaluation, where we train a simple model with the resulting encodings. Then in Section 4.7.2 we discuss the evaluation of the implicit approach based on autoencoders. We discuss the decoding accuracy, performance on logical properties discussed previously, and the theorem proving task of premise selection.

4.7.1 Experiments and Evaluation of Explicit Approach

We will present an evaluation of the explicit encoding models. First, we consider the properties the models have been trained with (cf. Section 4.3). Here, we have two different ways of obtaining evaluation and test data. We also want the encoding networks to generalize to, and preserve properties that it has not specifically been trained on. Therefore, we encode a set of formulas and expressions and train an SVM (without kernel modifications) with different properties on them.

[2]A more detailed description of the dataset can be found here: `https://github.com/JUrban/deepmath`

For the first and more straightforward evaluation, we use the data extracted dataset from the Graph Theory and Set Theory library described in Section 4.6.1 as training data. One could split this data before training into a training set and evaluation set so that the network is evaluated on unseen data. In this approach, however, constants, formulas, etc. occurring in the evaluation data may have been seen before in different contexts. For example, considering the Set Theory library, terms and formulas containing `union(X,Y)`, `intersection(X,Y)`, etc. will occur in training data and evaluation data. Indeed, in applications such as premise selection, such similarities and connections are actually desired, which is one of the reasons we use character-level encodings. Nevertheless, we will focus on more difficult evaluation/test data. We will use data extracted from the Category Theory library as evaluation data and the Set/Graph Theory data for training. Hence, training and evaluation sets are significantly different and share almost no terms, constants, formulas, etc. We train the models on embedding dimensions 32, 64, and 128 (we only consider 64 for projective models). The input length, i.e. the length of the formulas was fixed to 256 since this includes almost all training examples. The CNN models had 8 convolution/pooling layer pairs of increasing filter sizes (1 to 128), while the LSTM models consisted of 3 bidirectional LSTM layers each of dimension 256. In the "Fully Connected"-models we append two additional dense layers. Similarly, for the projective models, we append a dense layer with a lower output dimension.

The evaluation results of the models are shown in Table 4.1.

The multi-label subformula classification is not relevant for this evaluation since training and testing data are significantly different. However, the binary subformula classification is useful and proves to be a difficult property to learn[3]. Surprisingly, adding further fully connected layers seems to have no major effect for this property regardless of the underlying model. In contrast, the additional dense layers vastly improve the accuracy of the modus ponens classifier (from 49% to 97% for the simple CNN based model with embedding dimension 32). It does not make a difference whether these dense layers are projective or not. Interestingly, every LSTM model even the ones with dense layers fail when classifying this property. Similar observations although with a smaller difference can be made with the term-formula distinction. Classifying whether two terms are unifiable or not seems to be a task where LSTMs perform better. Generally, the results for unifiability are similarly good across models. When determining whether a formula is well-formed, CNN based models again outperform LSTMs by a long shot. In addition, a big difference in performance can be seen between CNN models with additional layers (projective or not) appended. Unsurprisingly alpha equivalence is a difficult property to learn especially for CNNs. This is the only property where LSTMs clearly outperform the CNN models. Thus combining LSTM and CNN layers into a hybrid model might prove beneficial in future works. In addition, having fully connected layers appears to be necessary in order to achieve accuracies significantly above 50%.

Generally, varying embedding dimensions does not seem to have a great impact on the performance of a model, regardless of the considered property. As expected, adding

[3]The binary subformula classification describes the following problem: Given two formulas, decide if one is a subformula of the other.

Network	embedding dimension	subformula multi-label classification	binary subformula classification	modus ponens	term vs formula classification	unifiability	well-formedness	alpha equivalence
CNN	32	0.999	0.625	0.495	0.837	0.858	0.528	0.498
CNN	64	0.999	0.635	0.585	0.87	0.73	0.502	0.55
CNN	128	0.999	0.59	0.488	0.913	0.815	0.465	0.587
CNN with Projection to 32	64	1.0	0.662	0.992	0.948	0.81	0.748	0.515
CNN with Projection to 64	64	1.0	0.653	0.985	0.942	0.718	0.85	0.503
CNN with Fully Connected layer	32	0.999	0.64	0.977	0.968	0.78	0.762	0.5
CNN with Fully Connected layer	64	0.999	0.668	0.975	0.973	0.79	0.77	0.548
CNN with Fully Connected layer	128	0.999	0.635	0.923	0.972	0.828	0.803	0.472
CNN with Fully Connected layer Pr to 32	64	1.0	0.648	0.973	0.922	0.865	0.69	0.487
CNN with Fully Connected layer Pr to 64	64	1.0	0.662	0.968	0.967	0.898	0.762	0.497
LSTM	32	1.0	0.652	0.488	0.975	0.883	0.538	0.508
LSTM	64	0.999	0.652	0.49	0.942	0.86	0.49	0.575
LSTM	128	1.0	0.643	0.473	0.96	0.885	0.51	0.467
LSTM Pr to 32	64	1.0	0.69	0.537	0.863	0.87	0.513	0.62
LSTM Pr to 64	64	1.0	0.598	0.535	0.845	0.902	0.515	0.575
LSTM with Fully Connected layer	32	0.999	0.638	0.485	0.855	0.902	0.532	0.692
LSTM with Fully Connected layer	64	1.0	0.63	0.491	0.882	0.848	0.52	0.833
LSTM with Fully Connected layer	128	1.0	0.635	0.473	0.968	0.887	0.51	0.715
LSTM with Fully Connected layer Pr to 32	64	1.0	0.657	0.495	0.96	0.883	0.505	0.672
LSTM with Fully Connected layer Pr to 64	64	1.0	0.62	0.503	0.712	0.898	0.492	0.662

Table 4.1: Accuracies of classifiers working on different encoding/embedding models. The models were trained on the Graph/Set theory data set and the evaluation was done on the unseen Category Theory data set. The LSTM based models are in grey. (Pr = Projection)

additional fully connected layers has no negative effect. This leads us to distinguish two types of properties: Properties where additional dense layers have a big impact on the results (modus-ponens, well-formedness, alpha-equivalence), and those where the effect of additional layers is not significant (unifiability, term-formula, bin. subformula). It does not seem to make a big difference whether the appended dense layers are projective or not. Even the embedding models that embed the formulas to an 8^{th} of the input dimension perform very well. Another way of classifying the properties is to group properties where CNNs perform significantly better (modus-ponens, well-formedness), and conversely where LSTMs are preferable (alpha equivalence).

Alternative Problems and Properties We also want the encodings of formulas to retain information about the original formulas and properties that the networks have not specifically been trained on. We want the networks to learn and preserve unseen structures and relations. We conduct two lightweight tests for this. First, we train simple models such as SVMs to recognize certain structural properties such as the existence of certain quantifiers, connectives, etc. (that we did not specifically train for) in the encodings of formulas. To this end, we train SVMs to detect logical connectives such as conjunction, disjunction, implication, etc. These classifications are important since logical connectives were not specifically used to train the encoding networks but are important nevertheless. Here, the SVMs correctly predict the presence of conjunctions, etc. with an accuracy of 85%. We also train an ordinary linear regression model to predict the number of occurring universal and existential quantifiers in the formulas. This regression correctly predicts the number of quantifiers with an accuracy of 94% (after rounding to the closest integer). These results were achieved by using the CNN based model with fully connected layers. We also evaluated the projective models with this method. We achieved 70% and 84% for classification and regression respectively using the CNN model with a fully connected and a projection layer. When using models that were trained using single layer classifiers as discussed in Section 4.5.1 we get better results for simple properties such as the presence of a conjunction.

4.7.2 Experiments and Evaluation of Implicit Approach

We also evaluate the encoding models based on the autoencoder setup. In our experiments, we first learn from unlabelled data. Hence, we take the entire dataset and discard all labels and simply treat them as formulas. Using this dataset we train encoders and decoders in 100k optimization steps. First we evaluate how a simple feed-forward network performs when tasked with classifying formulas based on their embeddings. To this end, we train a feed-forward network to classify input vectors according to properties given in the dataset (logical properties or whether the premise is useful in proving the conjecture). Those input vectors are given by an encoder network whose weights are frozen during this training. The classifier networks have 6 layers each with size 128 and nonlinear ReLU activation functions. Since the classification tasks for some properties require two formulas, the input of those classifiers is the concatenated encoding of the input formulas. We split the classification datasets into training, validation and test sets randomly, in

proportions 8-1-1. Every thousand optimization steps we evaluate the validation loss (the loss on the validation set) and report test accuracy from the lowest validation loss point during training.

Hyperparameters All autoencoding models were trained for 100k steps, using the Adam optimizer [KB14] with learning rate $1e-4$, $\beta_1 = 0.9, \beta_2 = 0.999, \epsilon = 1e-8$. All models work with 128 dimensional sequence token embeddings, and the dimensionality of the final formula encoding was also 128. All models (except for LSTM) are comprised of 6 layers. In the convolutional network after every convolutional layer, we apply maximum pooling of 2 neighboring cells. In the Transformer encoder, we use 8 attention heads. The autoencoders were trained for 100k optimization steps and the classifiers for 30k steps. The batch size was 32 for difference training, 16 for recursive training, and 32 for classifier networks.

Decoding accuracy

		Difference tr.		Recursive tr.	
		Formula	Symbol	Formula	Symbol
Mizar40 dataset	Convolutional	0.000	0.226	0.005	0.658
	WaveNet	0.000	0.197	0.006	0.657
	LSTM	0.000	0.267	0.063	0.738
	Transformer	0.000	0.290	0.006	0.691
Logical properties dataset	Convolutional	0.440	0.750	0.886	0.984
	WaveNet	0.420	0.729	0.865	0.981
	LSTM	0.451	0.759	0.875	0.979
	Transformer	0.474	0.781	0.916	0.990

Table 4.2: Decoding accuracy of tested encoders. "Formula" indicates the share of formulas successfully decoded. "Symbol" is the average amount of correctly decoded symbols in a formula.

After training the autoencoders (Figure 4.11 on page 55) using the unlabelled datasets we test their accuracy. That is, we determine how well the decoder can retrieve the original formulas. This is done recursively. First, the formula is encoded, then its top symbol is determined by the top symbol classifier and encodings of its subformulae are determined using subtree extractors. Then top symbols of those subformulae are found and so on. The results are presented in Table 4.2 on page 53. From the table, it is clear that the recursive training outperforms the difference training regardless of the encoding model or dataset. This result is not unexpected as the design of the recursive training is more considerate of the subformulas (i.e. subtrees). Hence, a wrong subtree prediction has a larger impact in the loss of the recursive training than in the difference training. Figure 4.10 on page 54 shows a plot of the decoding accuracy as the depth of the formula increases. Unsurprisingly, for very shallow formulas both types of networks

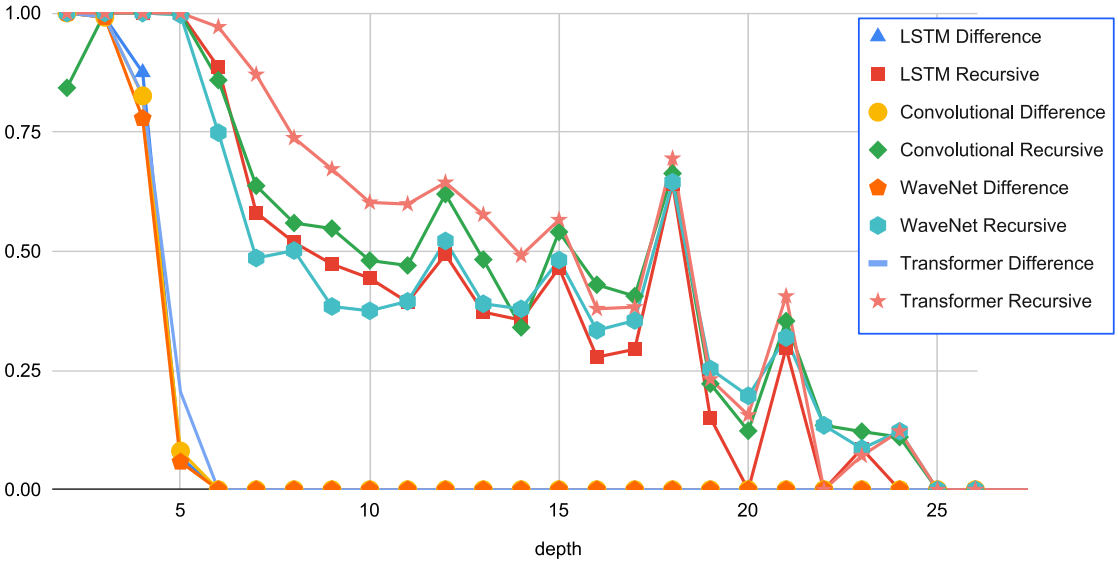


Figure 4.10: Decoding accuracy of formulas over formula depth in the logical properties dataset.

perform comparably, with the difference training accuracy dropping to almost zero as the formulas reach depths 5. On the other hand, the recursive models can almost perfectly recover formulas up to depth 5, which was our goal.

Logical properties

We also test if the encodings preserve logical properties presented in Section 4.6.1. In theory, this information still has to be present in some shape or form, but we want to test whether a commonly used feed-forward network can learn to extract them.

The results are shown in Table 4.3 on page 56. Comparing the models we notice a surprising result. Indeed, for some properties, the difference training performs on-par or even better than recursive training. This stands in contrast to the decoding accuracy presented previously where the recursive training outperforms the difference training across the board. This is likely due to the fact that some of the properties can be decided based only on the small top part of the tree, which the difference training does learn successfully (see Figure 4.10 on page 54).

Premise selection

As described before premise selection is an important task in interactive and automated theorem proving. We test the performance of our encodings for the task of premise selection on the Mizar40 dataset (described in Section 4.6.2). The experiment (as described in Section 4.7.2) involves first training the encoder layer to create formula embeddings, then training a feed-forward network to classify formulas by their usefulness in constructing a proof. The results are shown in Table 4.4 on page 57. Our general

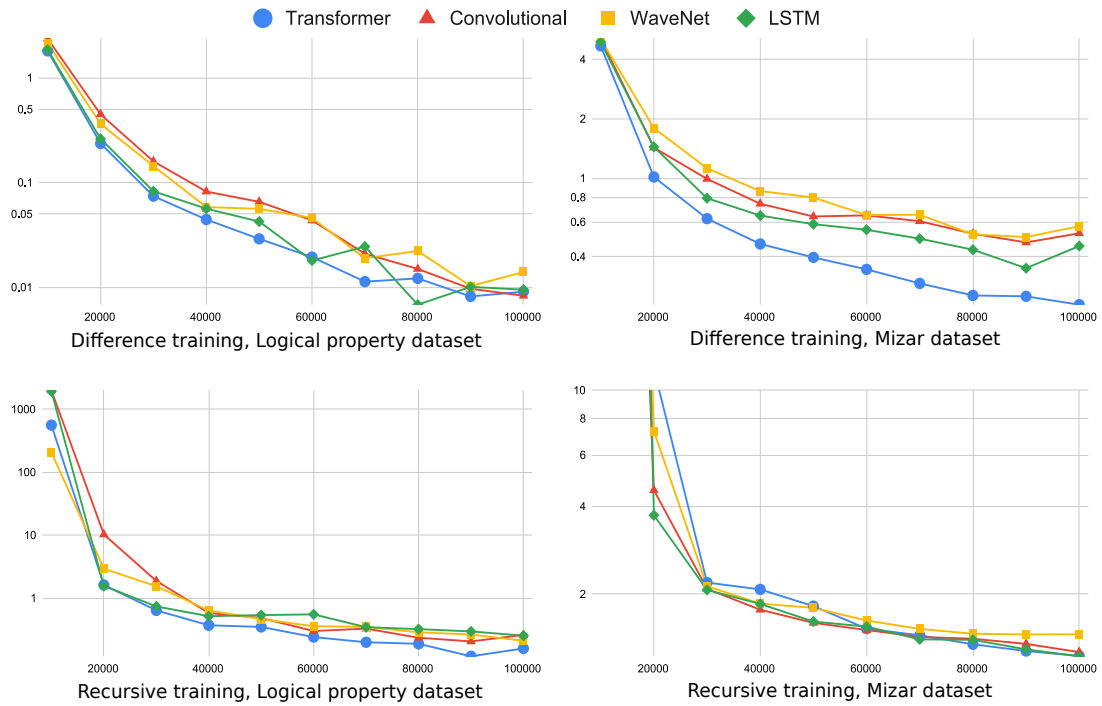


Figure 4.11: Loss during training. The top two graphs present loss during difference training, and the bottom two graphs during recursive training. Note a different vertical scale for the four graphs, this is because the losses for the different training modes and datasets are hard to compare, however, all four converge well.

decodable embeddings are better than the non-neural machine learning models, albeit perform slightly worse than the best classifiers currently in literature (81%) [CAC+19] (Which are non-decodable and single-purpose).

4.8 Conclusion

We have developed and compared logical formula encodings (embedding) inspired by the way human mathematicians work. The formulas are represented in an approximate way, namely as dense continuous vectors. The representations additionally allow for the application of reasoning steps as well as the reconstruction of the original symbolic expression (i.e. formula) that the vector is supposed to represent. The explicit approach enforces a number of properties that we would like the embedding to preserve. For example, basic structural properties (subformula property, etc) can be recovered, natural deduction reasoning steps can be recognized, or even unifiability between formulas can be checked (although with less precision) in the embedding. In the second approach, we propose to autoencode logical formulas. Here, we want the encoding of formulas

		Difference tr.	Recursive tr.
Subformula	Convolutional	0.736	0.870
	WaveNet	0.787	0.877
	LSTM	0.755	0.891
	Transformer	0.711	0.923
Modus Ponens	Convolutional	0.920	0.893
	WaveNet	0.903	0.866
	LSTM	0.941	0.916
	Transformer	0.498	0.946
Term vs Formula	Convolutional	1.000	0.979
	WaveNet	1.000	0.990
	LSTM	1.000	0.995
	Transformer	1.000	1.000
Unifiability	Convolutional	0.988	0.975
	WaveNet	0.991	0.991
	LSTM	0.990	0.990
	Transformer	0.989	0.990
Well-formedness	Convolutional	0.969	0.988
	WaveNet	1.000	0.992
	LSTM	1.000	1.000
	Transformer	0.996	0.996
Alpha equivalence	Convolutional	0.998	0.998
	WaveNet	0.998	1.000
	LSTM	0.483	1.000
	Transformer	0.990	1.000

Table 4.3: Logical property classification accuracy on test set.

to preserve enough information so that the encoded symbolic expression (formula) can be recovered from the embedding alone. As such sufficient information for the same logical and structural operations must be present. In addition, this also allows the actual computation of results of the inference steps or unifiers. We considered two different training setups for the autoencoders. One is called difference training and the other is recursive training. In order to train and evaluate the approaches, we developed several logical property datasets transformed from subsets of the TPTP problem set.

Apart from an evaluation on the TPTP dataset, we also evaluated the approaches on premise selection problems originating from the whole Mizar Mathematical Library. As expected, both difference and recursive training are less performant on the Mizar 40 dataset than on the logical properties dataset. We know of two reasons for this. First, the Mizar dataset is much bigger, both when it comes to the number of constants, types, but also the number of formulas and their average sizes. As such, fitting all the formulas in vectors of the same size is going to be less precise. Second, the formulas in the Mizar dataset are more uniformly distributed. As we use models with the same numbers and

	Difference tr.	Recursive tr.
Convolutional	0.681	0.696
WaveNet	0.676	0.696
LSTM	0.665	0.703
Transformer	0.670	0.704

Table 4.4: Premise selection accuracy on test set.

sizes of layers, memorizing parts of the Mizar dataset is clearly a more complex task. Despite these problems, the results are promising for both the formula reconstruction task and the original theorem proving tasks like premise selection.

The code of our embedding, the dataset, and the experiments are available at:

`http://cl-informatik.uibk.ac.at/users/cek/logcom2020/`

Future work could include considering further logical models and their variants. We have so far focused on first-order logic, however, it is possible to do the same for simple type theory or even more complex variants of type theory. This would allow us to do the premise selection analysis presented in this work for the libraries of more proof assistants. Finally, the newly developed capability to decode an embedding of a first-order formula could also be a useful technique to consider for conjecturing [GKU16] or proof theory exploration [CJRS13]. Finally, we imagine that then a reversible encoding of logical formulas could improve the proof guidance of first-order logic theorem provers.

Acknowledgements

This work has been supported by the ERC starting grant no. 714034 *SMART*.

Chapter 5

Adversarial Learning to Reason in an Arbitrary Logic

5.1 Abstract

Existing approaches to learning to prove theorems focus on particular logics and datasets. In this work, we propose Monte-Carlo simulations guided by reinforcement learning that can work in an arbitrarily specified logic, without any human knowledge or set of problems. Since the algorithm does not need any training dataset, it is able to learn to work with any logical foundation, even when there is no body of proofs or even conjectures available. We practically demonstrate the feasibility of the approach in multiple logical systems. The approach is stronger than training on randomly generated data but weaker than the approaches trained on tailored axiom and conjecture sets. It however allows us to apply machine learning to automated theorem proving for many logics, where no such attempts have been tried to date, such as intuitionistic logic or linear logic.

5.2 Introduction

In the last decade for many logical systems machine learning approaches have managed to improve on the best human heuristics. This worked well for example in classical first-order logic guiding the superposition calculus [JCO+20], tableaux calculus [OKU20], or even in higher-order logic [FB16]. In all these works strategies based on machine learning can significantly improve on the best human-designed ones.

To train such machine-learned strategies, datasets of problems and baselines on these problems are required. In particular successful proofs (and in some cases also unsuccessful proofs [KUMO18]) are gathered and used to train a machine-learned version of the prover.

In this work, we consider the same problem but without a fixed logic and without a dataset of problems given. We apply a policy-guidance algorithm (known for example from AlphaZero [SHS+17]) to proving in an arbitrary logic without a given problem set. In particular we:

- propose a theorem-construction game that allows for learning theorem proving with AlphaZero, without relying on training data;

- propose the first dataset for learning for multiple logics, together with learning baselines for this dataset.
- propose an adjusted Monte-Carlo Tree Search that is able to take into account certain (sure) information, when a player makes multiple moves (explained in "Certain Value Propagation" section);
- evaluate the trained prover on the dataset showing that it improves proof capability in various considered logics; and
- as many other problems and games can be directly encoded as logical problems we show that the proposed universal learning for logic also works on some encoded games, such as Sokoban.

5.3 Related Work

[SHS+17] have shown that Monte-Carlo tree search combined with reinforcement learning applied to policy and value functions can generalize to multiple logical games (Go, Chess, Shogi).

The work of [FAA+21] focuses on classical first-order logic without equality and the resolution calculus (so only one of the many logics we consider) and (like us) does not use the complete TPTP problems, but only the axiom sets to learn. There is however a significant overlap between the axioms and conjectures in other problems. The resulting prover does learn to prove theorems but is significantly weaker than E-prover on the TPTP problems. The idea to use only the axioms has already been considered by [BLRS19]. Even if no original conjectures are exposed to the prover, the dataset used for training is quite large, in comparison with ours, where no formulas are given at all.

As already discussed in the introduction, there are many approaches to applying MCTS with policy and guidance learning in various fixed calculi and on fixed datasets [KUMO18, RR19]. The results are better than those we are able to get here, but no new logics or problems are tried and generalization and transfer have been very limited so far. The AlphaZero algorithm has also been applied in theorem proving to the synthesis of formulas [BG19] and functions [Gau20].

Kaiser et al. demonstrated that theorem proving can be used to solve many games, such as the ones in the General Game Playing competition [KS11]. With the current work, we show that learning for logic can be also applied to these games.

5.4 Preliminaries

5.4.1 AlphaZero

The core of the AlphaZero algorithm [SHS+17] is learning from self-play. It trains a neural network to evaluate the states of a game to estimate the final outcome of the game as well as a policy maximizing the expected outcome. Using the neural network in

the current stage of learning a lot of playouts are generated, then this data is used as training data for further improvement.

For training value estimation, the algorithm uses the actual outcomes of the games. To train policy estimation, a Monte-Carlo Tree Search (MCTS) is used to compute a better policy, then the network is trained to return this better policy.

5.4.2 Monte-Carlo Tree Search

To train the policy evaluating network, we need to provide it with a somewhat better policy. This is done by exploring a tree of possible moves. It is a guided exploration, biased towards the moves pointed to by the policy and to where the value estimations are higher.

A tree is constructed with every node representing a state of the game. Each of those states is evaluated using the neural network. When deciding where to add a new node (thus exploring a branch of the game further) the MCTS algorithm takes into account both the value and the policy estimations from the neural network (biased toward following the policy and higher values), as well as how well a branch was already explored (biased to explore yet unexplored branches more).

After adding a set amount of nodes to the tree, the new better policy is defined to be proportional to the number of nodes explored below each of the immediate children of the root node (representing the state for which we are computing a better policy).

5.5 Approach

5.5.1 The theorem-construction game

We propose a two-player game such that a system trained to play the game well could be used to effectively prove the theorems of a given logic system. The first player (referred to as *adversary*) constructs a provable statement, while the other player (referred to as *prover*) tries to prove it. The goal of the adversary is to construct such a theorem, that the prover will fail to prove it. However, because of the available game moves (construction steps), the statements are always provable.

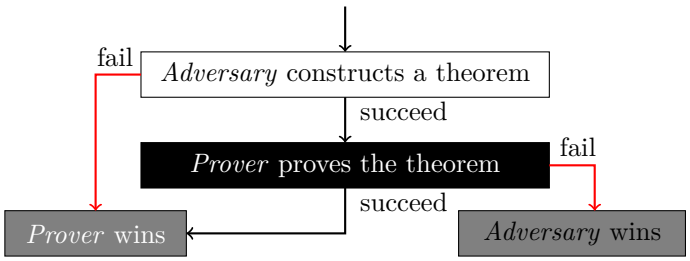


Figure 5.1: High-level overview of the theorem-construction game.

We represent the game objects using Prolog-like *terms*, where a term can be either a *variable* or a pair of an atom and a list of subterms. In the examples, we use the

convention of marking variables with capital letters, and denoting compound terms as an atom name followed by a list of subterms in brackets (skipped when the list is empty). For example $\texttt{tee}(A, \texttt{implies}(B, \texttt{false}))$, which can also be expressed with operators like $(A \vdash (B \to \bot))$.

The construction game is defined for a given set of *inference rules*. An inference rule is a pair of a term and a list of terms, that can share variables. For example $\texttt{tee}(A, \texttt{and}(B, C)) \leftarrow \texttt{tee}(A, B), \texttt{tee}(A, C)$, equivalent to $(A \vdash (B \wedge C)) \leftarrow (A \vdash B), (A \vdash C)$.

For the prover, a game state consists of a list of terms that need to be proven (together with the information that the prover is making the move). During their move, a player can choose one of the given inference rules (the *action space* is the set of *inference rules*), and apply it to the first term of the list. The left side of the rule is then unified with that term. If the unification fails, the player making the move loses. If it succeeds, the term is removed from the list, and the right side of the rule (after unification) is added.

The *adversary* (the player constructing a theorem) makes moves in much the same way, except instead of starting with a theorem to be proven, it starts with a single variable. Applying inference rules to prove this variable will unify it with some term. If the proof is successfully completed, this variable we started with will be unified with a provable theorem. We keep track of this variable in the game state. When the adversary finishes its proof, we pass the constructed theorem to the prover, after replacing all remaining variables with fresh constants.

The second player tries to prove the theorem, winning when the list is empty. To better illustrate the working of our theorem-construction game we present the rules of a concrete game in Figure 5.2 and an example playout in Figure 5.3.

$$A, B \vdash A \leftarrow \tag{5.1}$$
$$A \vdash (B \to C) \leftarrow (B, A \vdash C) \tag{5.2}$$
$$A \vdash (B \wedge C) \leftarrow (A \vdash B), (A \vdash C) \tag{5.3}$$
$$A \vdash B \leftarrow (A \vdash (B \wedge C)) \tag{5.4}$$
$$A \vdash B \leftarrow (A \vdash (C \wedge B)) \tag{5.5}$$
$$A \vdash B \leftarrow (A \vdash \bot) \tag{5.6}$$

Figure 5.2: A subset of propositional logic inference rules used in the example in figure 5.3

5.5.2 Certain Value Propagation

The AlphaZero [SHS+17] algorithm utilizes a neural network to estimate state values (a number in range $(-1, 1)$, we will call it v_θ) and policies (a vector with as many dimensions as the size of the action space). Then a Monte Carlo Tree Search (MCTS) [KS06] is used

rule	terms to be proven	constructed theorem
	X	X
2	$A, B \vdash C$	$B \vdash A \to C$
3	$(A, B \vdash D), (A, B \vdash E)$	$B \vdash A \to (D \wedge E)$
4	$(A, B \vdash (D \wedge F)), (A, B \vdash E)$	$B \vdash A \to (D \wedge E)$
1	$(D \wedge F), B \vdash E$	$B \vdash (D \wedge F) \to (D \wedge E)$
6	$(D \wedge F), B \vdash \bot$	$B \vdash (D \wedge F) \to (D \wedge E)$
5	$(D \wedge F), B \vdash (G \wedge \bot)$	$B \vdash (D \wedge F) \to (D \wedge E)$
1		$B \vdash (D \wedge \bot) \to (D \wedge E)$
	$\mathtt{b} \vdash (\mathtt{d} \wedge \bot) \to (\mathtt{d} \wedge \mathtt{e})$	
2	$(\mathtt{d} \wedge \bot), \mathtt{b} \vdash (\mathtt{d} \wedge \mathtt{e})$	
6	$(\mathtt{d} \wedge \bot), \mathtt{b} \vdash \bot$	
5	$(\mathtt{d} \wedge \bot), \mathtt{b} \vdash (A \wedge \bot)$	
1	Prover won	

Figure 5.3: An example of a playout of the theorem-construction game with the inference rules shown in figure 5.2

to compute better estimates of value and policy. This is done by exploring the tree of possible playouts, with a bias toward where value and policy lead to.

During this exploration, MCTS maintains a better value estimation of every state (we will refer to it as v), which is defined to be the average of v_θ of all explored descendants. We will use an equivalent definition, as the weighted average of immediate children, with weights being the number of visits of a given node (the difference will become important).

$$v(n) = \frac{v_\theta(n) + \sum_{d<n} v_\theta(d)}{|d : d < n| + 1} = \\ = \frac{v_\theta(n) + \sum_{c \in \text{children}(n)} v(c) * (|d : d < c| + 1)}{|d : d < n| + 1}$$

In our version of MCTS, for every node, we keep track of a lower and upper bound for possible node values. For non-final nodes, these are simply $(-1, 1)$ (as this is the range of possible outcomes), but for the final nodes, the bounds are both equal to the final reward. These bounds are propagated up the tree in a natural way (taking into account state ownership). Then, for every node we compute a new v_c value, which is simply v adjusted to fall within lower-upper bounds – so eg. if the lower bound is higher than v, then v_c will be equal to the lower bound. Then, when computing v for the nodes above we use this new v_c value rather than the old v.

$$v_c(n) = \max(\text{lower}(n), \min(\text{upper}(n), v(n))) \\ v(n) = \frac{v_\theta(n) + \sum_{c \in \text{children}(n)} v_c(c) * (|d : d < c| + 1)}{|d : d < n| + 1}$$

.

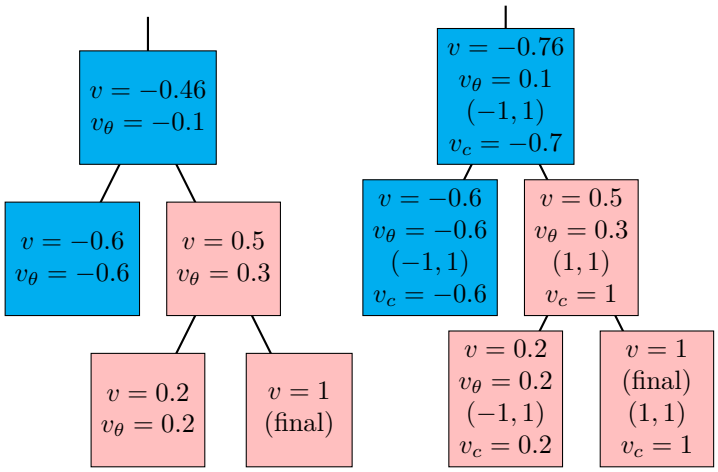


Figure 5.4: Examples of value estimation in MCTS without Certain Value Propagation (left) and with (right). State ownership (which player is making a move) is marked with color.

Additionally, whenever the value estimation of a state is determined to be -1 (the lowest possible outcome), this state will be avoided. This avoidance is applied both to MCTS exploration and the choice of an action to take during playouts.

The impact of certain value propagation on the final performance of the prover is shown in Fig. 5.5.

5.5.3 Auxiliary replays

To facilitate the prover learning to prove theorems constructed by the adversary we add additional *auxiliary replays*. These come from the games won by the adversary when the prover fails to prove a constructed theorem. Because of the way the theorem was constructed we know how to prove it – we just need to apply the same moves that the adversary used to construct it. Using this fact, we create a replay that shows how the theorem could be proven. In this replay, the policy is not computed using MCTS, but rather is just a one-hot vector pointing to the move that the adversary made when constructing the theorem.

With these auxiliary replays, our algorithm can be considered to train on artificially constructed theorems, that at first come from simply randomly applying inference rules, but later on uses neural guidance to find theorems that the prover cannot yet prove. However, as mentioned earlier, we only do this for theorems that the prover failed to prove.

The impact of including auxiliary replays on the final performance of the prover is shown in figure 5.5.

5.5.4 Balancing training data

Since the theorem proving game (explain in section "The theorem-construction game") is asymmetrical, simply using all replay data for training would result in an unbalanced dataset. On top of this, we use auxiliary replays (explained in section "Auxiliary replays"), further disturbing the training data.

To deal with this imbalance we apply training data balancing. Replays are split into parts according to which player won, and the third set of auxiliary replays. All training batches contain the same number of examples from each part.

However, this means disturbing the way the Mean Square Error loss works for value estimation. Consider a value estimation of the starting state. Normally, optimal loss for it would be achieved if the estimate was the average outcome of the game, but with balancing the optimal loss will be achieved by estimating the value to be 0 (mean between losing and winning). This problem affects every state that occurs multiple times in the training dataset.

To counteract this problem, the value loss is weighted in proportion to the size of the part of the data, from which the point originates. So if a player A won in proportion $4 : 1$, the training batches would include games won by this player in proportion $1 : 1$, but $\frac{4}{5}$ of the loss (and therefore gradients) would be determined by data from games won by the player A. For auxiliary replays, this weight is set to 0 and only policy is learned from them.

The impact of balancing training data and weighing the value loss on the final performance of the prover is shown in figure 5.5.

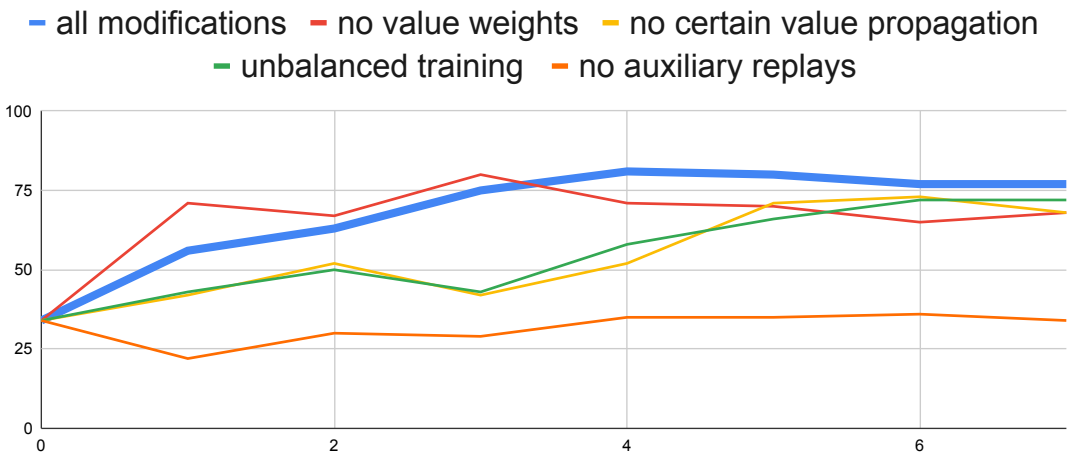


Figure 5.5: The impact of our modifications on our algorithm. Solved first-order classical tableaux test problems over time (episodes) with 5k games per episode.

5.5.5 Applicability

As mentioned in section "The theorem-construction game", our algorithm works with a logic system defined by a set of inference rules. This set of rules can be thought of as a Prolog program, and since Prolog is Turing-complete our method can (at least theoretically) be used to learn to reason in any formally defined (and decidable) context. As an example of wide applicability, we train our system to solve Sokoban puzzles. This can be done by defining rules of the game as inference rules of a pseudo-"logic system".

This of course does not mean that the system will always work well. For example, a saturation prover requires all terms used in the proof to be fully determined from the start. This negates the advantage of the adversary player, who normally can still modify the constructed theorem late into its proof. Because of this, the probability of constructing a theorem that an untrained prover cannot prove becomes really low – so low that potentially no such theorems will be generated for the initial training set. In such a situation, nothing can be learned from such data and the system is stuck. This problem could potentially be overcome by simply generating enough playouts, but in our experiments with using a saturation prover, the system got stuck after the first step, with the prover winning all games. This was the case in our experiments using a saturation proving method, with 10^4 games generated per episode (possibly more games could help).

5.5.6 Failure states

Another consequence of the game being asymmetrical is the possibility of the training getting stuck when one player starts winning every time. The mechanism of auxiliary replays mentioned above counteracts this to some extent, allowing the prover to still learn even if the adversary is always successfully constructing a hard enough theorem. If the prover was winning every game, however, we would need to rely on exploration for the adversary to find something hard to prove. This situation is however virtually impossible, because of the exploration noise used during playouts. This should lead to adversary towards theorems where the prover is uncertain and sometimes loses due to exploration noise, and then to theorems where the prover fails.

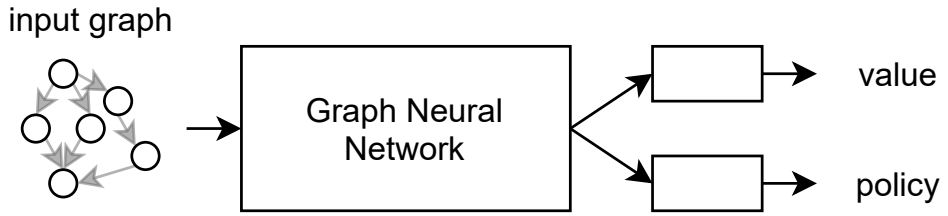
There is however another failure state which if reached would be entirely stable. It is possible because the construction of theorems is inherently easier than proving. Consider a theorem $\exists_x \text{HASH}(x) = y$. It is easy to prove such a theorem if one can choose what y is going to be. If y is already decided, proving such a theorem becomes extremely difficult. So difficult in fact, that we cannot hope that a neural network would be able to learn to do this.

If the adversary found such a space of *Uninteresting Hard Theorems*, it would never learn to do anything else. After all, it is a winning strategy for this game. The prover, even using the auxiliary replays would never learn to do anything useful in this situation, and would gradually forget all the useful knowledge learned previously.

This does not seem to happen in any of our experiments. In some of our considered logic systems, it is not even clear that such an Uninteresting Hard Theorem space exists.

5.5.7 Neural architecture

For evaluating state value and policy we use a Graph Neural Network similar to the one described in [Pur20]. It is essentially a Graph Attention Network [VCC+17] using dot-product attention from the Transformer model [VSP+17] with different attention masks for different attention heads. One Graph Neural Network is used to create a single vector representation of the graph, which is then fed to the final layers to estimate policy and value.



Game states are represented as syntactic graphs. One graph contains all terms that need to be proven, together with information about which player the state belongs to, and (for the adversary player) the state of the constructed theorem. An example of such a graph is shown in figure 5.6.

A single Graph Neural Network is used to evaluate the states for both players, the prover and the adversary.

5.6 Evaluation

To test the impact of our method of adversarial training we compare an algorithm trained using our theorem-construction game with a prover trained using uniformly generated random data (an approach somewhat similar to [FAA+21]).

The methods are tested on a dataset not seen by either approach. This test dataset is human-generated (see section "Considered logics" for details on each test dataset) and is often very far outside the training distribution.

5.6.1 Baseline

We generate baseline training data by applying inference rules randomly. This is essentially the adversary from our game doing random moves. Because this does not require evaluating states with a neural network, generating such data is much cheaper, so we generate more playouts – 10^6 (we note that not all playouts result in a constructed theorem).

We use all data generated this way to train a network to estimate policy and value. The policy is a one-hot vector pointing to what the adversary did to construct a theorem, and the value is 0.99^n with n being the number of moves left to do.

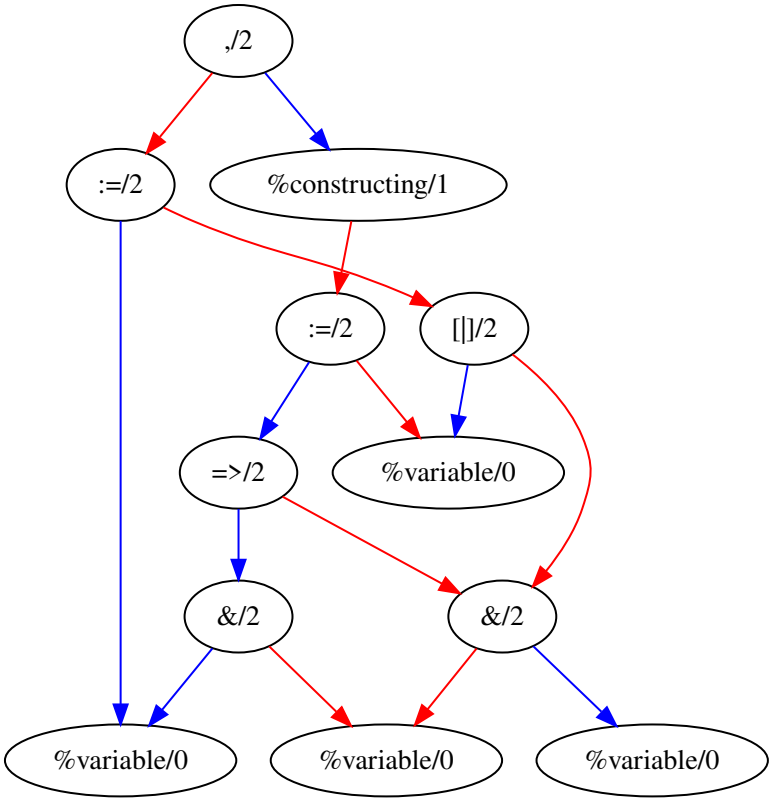


Figure 5.6: An example graph representing the game state from the game in figure 5.3 after initial 4 moves.

5.6.2 Setup

We implemented the proposed theorem-construction game engine SWI-Prolog [WSTL10] using PyTorch [PGM+19] for the proposed adversarial neural architecture.

We train our system in episodes, first generating 10^4 playouts, then training the neural network using these playouts as training data. This step is repeated multiple times, and after every one, we evaluate the system using the test dataset.

5.6.3 Experiments

To test how much the prover has learned, we play the game in a similar way to when training, except skipping the construction phase, and instead using a theorem from the test set. During such testing we forgo forcing exploration – we do not add exploration noise in Monte-Carlo Tree Search and use the most probable action instead of choosing randomly. Also, when a final state is found during MCTS exploration, we just follow a path to it.

For termination during testing, we use a limit on explored states – nodes added to the MCTS tree. Because a part of the tree can be reused for the next state (the part below

logic	baseline	best during game training	total solved during training
int. prop. sequent	12	12	**13**
classical FO sequent	**42**	39	40
classical FO tableaux	73	79	**83**
classical FO Hilbert	**38**	37	**38**
modal K prop. sequent	2	5	**7**
modal T prop. sequent	6	12	**13**
modal S4 prop. sequent	2	6	**8**
modal S5 prop. sequent	4	24	**24**
linear prop. sequent	37	34	**39**
sokoban solving	4	10	**12**

Table 5.1: Results of training in all tested logics – *int.* and *prop.* stand for intuitionistic and propositional

the node that was chosen) this does not imply any strict turn limit.

5.6.4 Considered logics

Intuitionistic We train our prover on sequent calculus in propositional intuitionistic logic [HR00]. For test theorems, we use a part of the ILTP library [ROK05].

Classical We run three experiments with classical first-order logic, trying out three different proof systems. One is sequential calculus, the same as used with intuitionistic logic, another is the Tableaux connection prover [Häh01], and lastly the rather unwieldy Hilbert system. For the test set, we use a small subset of the Mizar40 dataset [KU15] of formulas that do not use equality.

Linear We also train in linear logic [Gir87], only in the propositional setting. For the evaluation we use the LLTP [OdPPR20] library, most of which is taken from ILLTP [OdPPR19]. We also use a few hand-written examples.

Modal In another experiment, we train the prover to work with modal propositional logic [BvBW07], in four variants: K, T, S4, S5. Each of those extends the definition of the logic by an additional rule.

For evaluation, we use the propositional part of the QMTLP library [RO11]. The set of test theorems is expanded for each consecutively added rule.

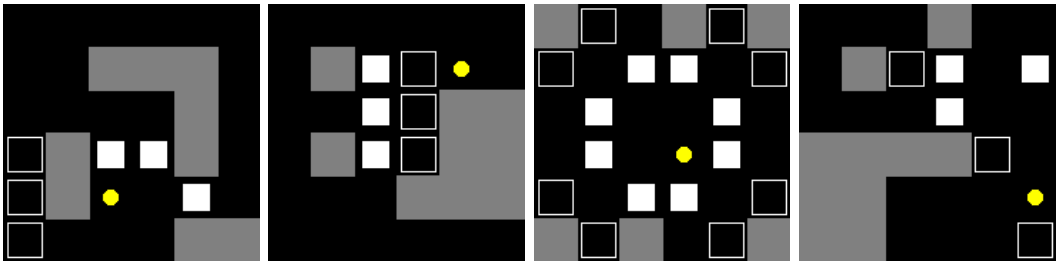


Figure 5.7: A few examples of Sokoban problems

Sokoban Sokoban is a classic puzzle game, where the goal is to push boxes into their target positions. The puzzle is PSPACE-complete [Cul97]. We only generate puzzles of size 6 by 6. For testing, we use a dataset available online[1] (only the problems that can fit into a 6 by 6 grid). A few examples of such problems are shown in figure 5.7.

5.7 Results and Discussion

The results of the evaluation are presented in table 1. We compare the baseline against the last model trained in the adversarial setup and additionally list the number of unique problems solved in all training epochs. For all modal logics as well as for classical first-order Tableaux the proposed adversarial theorem-construction game leads to many more solved problems. For a number of other logical calculi, the adversarial version is slower but leads to finding solutions different than those trained in the supervised setting, therefore leading to a large number of total solutions found. This is the case for intuitionistic sequent calculus and linear logic. Among the tried calculi, only for the classical sequent-calculus and Hilbert calculus, there is no advantage - this is likely due to the fact that the calculus is much closer to the syntax and the learned baseline can generalize enough. Finally, for the encoded Sokoban games the results are particularly good, with many games solved only in the adversarial-logical setting. We believe, that the adversary learns to construct more and more complex Sokoban-encoded proof games, while the player learns to solve them, in a way similar to curriculum learning.

«««< HEAD

5.7.1 Forwards vs. backwards conjecturing

During our experiments we briefly considered implementing a different method of constructing theorems, namely forward constructing: starting from assumptions, working towards the theorem. This method is used in [FAA+21] to generate synthetic data (though without any training for the generator).

The problem with forward constructing (and the reason we decided not to use it) can

[1] https://sourceforge.net/projects/sokoban-solver-statistics/

be illustrated using the following inference rule (disjunction elimination):

$$(T \vdash C) \leftarrow (T \vdash A \vee B), (T \vdash A \rightarrow C), (T \vdash B \rightarrow C)$$

«««< HEAD For the adversary to ever successfully apply such a rule, it would first have to construct three statements specifically fitting the rule. In the initial phase of random exploration that would be extremely unlikely. Moreover, the policy predictor will quickly learn that applying this rule always ends in failure, thus making its use even more unlikely after some training. In backward construction the rule can easily be applied, and simply results in three new statements that consequently need to be proven. ======= For the adversary to ever successfully apply such a rule, it would first have to construct three statements specifically fitting the rule. In the initial phase of random exploration that would be extremely unlikely. Moreover, the policy predictor will quickly learn that applying this rule always ends in failure, thus making its use even more unlikely after some training. In backwards construction the rule can easily be applied, and simply results in three new statements that consequently need to be proven. »»»> master

This problem essentially does not exists in the case of a saturation prover, which uses a single inference rule that can be applied anywhere.

5.7.2 Comparison with existing methods

Saturation provers are the state-of-the-art for first-order theorem proving are. These are designed narrow down the search space compared to possibly applying any inference rule at any point. Moreover, their solutions often involve millions of inference steps, while in our case the limit of moves is in the order of 10^2. As such many problems from the test dataset may not even technically be solvable by our system.

«««< HEAD For these reasons, our system performs a lot worse than these in their respective domains. It can however be applied to any formally defined domain and is (as far as the authors know) the only proposed theorem proving system that may continuously learn and improve without any dataset. ======= For these reasons, our system performs a lot worse than these in their respective domains. It can however be applied to any formally defined domain and is (as far as the authors know) the only proposed theorem-proving system that may continuously learn and improve without any dataset. »»»> master

5.8 Conclusions

We presented an algorithm for learning to reason in an arbitrary logic. The system, given only a formal definition of a logic, learns to construct increasingly harder problems in the logic and learns to prove them. We show that the system does learn to perform better than a baseline system trained using uniformly generated logical problems. The performance is of course weaker than that of domain-specific Automated Theorem Provers and provers trained on tailored datasets. We are, however, able to construct automatically the first efficient learned automated theorem provers for some logics where none existed before, including various modal logics.

Future work includes encoding more intricate theorem proving calculi, in order to compare them with the more tailored machine-learned systems. Furthermore, for most of the considered logics, the performance on the test sets has stagnated after a few episodes. It remains an open question if trying a compute power comparable with AlphaZero [SHS+17] would produce significantly better results.

Acknowledgements This work has been supported by the ERC starting grant no. 714034 SMART.

Chapter 6

Learning Higher-Order Logic Programs from Failures

6.1 Abstract

Learning complex programs through *inductive logic programming* (ILP) remains a formidable challenge. Existing higher-order enabled ILP systems show improved accuracy and learning performance, though remain hampered by the limitations of the underlying learning mechanism. Experimental results show that our extension of the versatile *Learning From Failures* paradigm by higher-order definitions significantly improves learning performance without the burdensome human guidance required by existing systems. Our theoretical framework captures a class of higher-order definitions preserving soundness of existing subsumption-based pruning methods.

6.2 Introduction

Inductive Logic Programming, abbreviated ILP, [Mug91, NCWSC97] is a form of symbolic machine learning which learns a logic program from background knowledge (BK) predicates and sets of positive and negative example runs of the goal program.

Naively, learning a logic program which takes a positive integer n and returns a list of list of the form $[[1], [1, 2], \cdots, [1, \cdots, n]]$ would not come across as a formidable learning task. A logic program is easily constructed using conventional higher-order (HO) definitions.

$$\begin{array}{|c|}\hline \texttt{allSeqN}(N, L)\texttt{:-}\ \texttt{iterate}(succ, 0, N, A),\ \texttt{map}(p, A, L). \\ \texttt{p}(A, B)\texttt{:-}\ \texttt{iterate}(succ, 0, A, B). \\ \hline \end{array}$$

The first `iterate`[1] produces the list $[1, \cdots, N]$ and `map` applies a functionally equivalent `iterate` to each member of $[1, \cdots, N]$, thus producing the desired outcome. However, this seemingly innocuous function requires 25 literals spread over five clauses when written

[1] See Appendix of arXiv:2112.14603 for HO definitions.

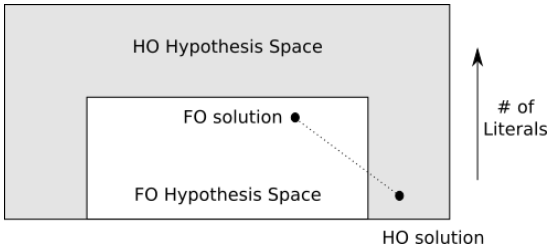


Figure 6.1: Inclusion of HO definitions increases the size of the search space, but can lead to the search space containing a shorter solutions.

as a function-free, first-order (FO) logic program, a formidable task for most if not all existing FO ILP approaches [CDEM22].

Excessively large *BK* can, in many cases, lead to performance loss [Cro20, SKB03]. In contrast, adding HO definitions increases the overall size of the search space, but may result in the presence of significantly simpler solutions (see Figure 6.1). Enabling a learner, with a strong bias toward short solutions, with the ability to use HO definitions can result in improved performance. We developed an HO-enabled *Popper* [CM21a] (*Hopper*), a novel ILP system designed to learn optimally short solutions. Experiments show significantly better performance on hard tasks when compared with *Popper* and the best performing HO-enabled ILP system, $Metagol_{\text{HO}}$ [CMM20]. See Section 6.5.

Existing HO-enabled ILP systems are based on *Meta-inter-pretive Learning* (MiL) [MLPT14]. The efficiency and performance of MiL-based systems is strongly dependent on significant human guidance in the form of *metarules* (a restricted form of HO horn clauses). Choosing these rules is an art in all but the simplest of cases. For example, `iterate`, being ternary (w.r.t. FO arguments), poses a challenge for some systems, and in the case of $HEXMIL_{\text{HO}}$ [CMM20], this definition cannot be considered as only binary definitions are allowed (w.r.t. FO arguments).

Limiting human participation when fine-tuning the search space is an essential step toward strong symbolic machine learning. The novel *Learning from Failures* (LFF) paradigm [CM21a], realized through *Popper*, prunes the search space as part of the learning process. Not only does this decrease human guidance, but it also removes limitations on the structure of HO definitions allowing us to further exploit the above-mentioned benefits.

Integrating HO concepts into MiL-based systems is quite seamless as HO definitions are essentially a special type of metarule. Thus, HO enabling MiL learners requires minimal change to the theoretical foundations. In the case of LFF learners, like *Popper*, the pruning mechanism influences which HO definitions may be soundly used (See page 813 of [CM21a]).

We avoid these soundness issues by indirectly adding HO definitions. *Hopper* uses FO instances of HO definitions each of which is associated with a set of unique predicates symbols denoting the HO arguments of the definition. These predicates symbols occur in the head literal of clauses occurring in the candidate program **iff** their associated FO instance occurs in the candidate program. Thus, only programs with matching structure

may be pruned. We further examine this issue in Section 6.4 and provide a construction encapsulating the accepted class of HO definitions.

Succinctly, we work within the class of HO definitions that are *monotone* with respect to *subsumption* and *entailment*; $p_1 \leq_\theta (\models)\ p_2 \Rightarrow H(p_1) \leq_\theta (\models)\ H(p_2)$ where p_1 and p_2 are logic programs, and $H(\cdot)$ is an HO definition incorporating parts of p_1 and p_2. Similar to classes considered in literature, our class excludes most cases of HO negation (see Section 6.4.4). However, our framework opens the opportunity to invent HO predicates during learning (an important open problem), though this remains too inefficient in practice and is left to future work.

6.3 Related Work

The authors of [CMM20] (Section 2) provide a literature survey concerning the synthesis of Higher-Order (HO) programs and, in particular, how existing ILP systems deal with HO constructions. We provide a brief summary of this survey and focus on introducing the state-of-the-art systems, namely, HO extensions of *Metagol* [CM16] and *HEXMIL* [KEI18]. Also, we introduce Popper [CM21a], the system *Hopper* is based on. For interested readers, a detailed survey of the current state of ILP research has recently been published [CDEM22].

6.3.1 Predicate Invention and HO Synthesis

Effective use of HO predicates is intimately connected to auxiliary Predicate Invention (PI). The following illustrates how `fold/4` can be used together with PI to provide a succinct program for reversing a list:

$$\begin{array}{|c|}\hline \texttt{reverse}(A,B)\texttt{:-}\ \texttt{empty}(C),\ \texttt{fold}(p,C,A,B). \\ \texttt{p}(A,B,C)\texttt{:-}\ \texttt{head}(C,B),\ \texttt{tail}(C,A). \\ \hline \end{array}$$

Including `p` in the background knowledge is unintuitive. It is reasonable to expect the synthesizer to produce it. Many of the well known, non-MiL based ILP frameworks do not support predicate invention, *Foil* [Qui90], *Progol* [Mug95], *Tilde* [Blo99], and *Aleph* [Sri01] to name a few. While there has been much interest, throughout ILP's long history, concerning PI, it remained an open problem discussed in "ILP turns 20" [MRP+12]. Since then, there have been a few successful approaches. Both *ILASP* [LRB14] and *δILP* [EG18] can, in a restricted sense, introduce invented predicates, however, neither handles infinite domains nor are suited for the task we are investigating, manipulation of trees and lists.

The best-performing systems with respect to the aforementioned tasks are *Metagol* [CM16] and *HEXMIL* [KEI18]; both are based on *Meta-interpretive Learning* (MiL) [MLPT14], where PI is considered at every step of program construction. However, a strong language bias is needed for an efficient search procedure. This language bias comes in the form of *Metarules* [CM14], a restricted form of HO horn clauses.

Definition 6.1 ([CT20])**.** A *metarule* is a second-order Horn clause of the form $A_0 \leftarrow A_1, \cdots, A_n$, where A_i is a literal $P(T_1, \cdots, T_m)$, s.t. P is either a predicate symbol or a HO variable and each T_i is either a constant or a FO variable.

For further discussion see Section 6.3.2. *Popper* [CM21a], does not directly support PI, though, it is possible to enforce PI through the language bias (*Poppi* is an PI-enabled extension [CM21b]). *Popper*'s language bias, while partially fixed, is essentially an arbitrary ASP program. The authors of [CM21a] illustrate this by providing ASP code emulating the chain metarule[2] (see Appendix A of [CM21a]). We exploit this feature to extend *Popper*, allowing it to construct programs containing instances of HO definitions. *Hopper*, our extension, has drastically improved performance when compared with *Popper*. *Hopper* also outperforms the state-of-the-art MiL-based ILP systems extended by HO definitions. For further discussion of *Popper* see Section 6.3.3, and for *Hopper* see Section 6.4.

6.3.2 *Metagol* and *HEXMIL*

We briefly summarize existing HO-capable ILP systems introduced by A. Cropper *et al.* [CMM20].

Higher-order *Metagol*

In short, *Metagol* is a MiL-learner implemented using a Prolog meta-interpreter. As input, *Metagol* takes a set of predicate declarations *PD* of the form `body_pred(P/n)`, sets of positive E^+ and negative E^- examples, *compiled* background knowledge BK_c, and a set of metarules M. The examples provide the arity and name of the goal predicate. Initially, *Metagol* attempts to satisfy E^+ using BK_c. If this fails, then *Metagol* attempts to unify the current goal atom with a metarule from $m \in M$. At this point *Metagol* tries to prove the body of metarule m. If successful, the derivation provides a Prolog program that can be tested on E^-. If the program entails some of E^-, *Metagol* backtracks and tries to find another program. Invented predicates are introduced while proving the body of a metarule when BK_c is not sufficient for the construction of a program.

The difference between *Metagol* and $Metagol_{\text{HO}}$ is the inclusion of *interpreted* background knowledge BK_{in}. For example, `map/3` as BK_{in} takes the form:

```
ibk([map,[],[],_],[]).
ibk([map,[A|As],[B|Bs],F],[[F,A,B],[map,As,Bs,F]]).
```

Metagol handles BK_{in} as it handles metarules. When used, *Metagol* attempts to prove the body of `map`, i.e. $F(A, B)$. Either F is substituted by a predicate contained in BK_c or replaced by an invented predicate that becomes the goal atom and is proven using metarules or BK_{in}.

A consequence of this approach is that substituting the goal atom by a predicate defined as BK_{in} cannot result in a derivation defining a Prolog program. Like with metarules,

[2] $\texttt{P}(A, B)$:- $\texttt{Q}(A, C), \texttt{R}(C, B)$.

additional proof steps are necessary. The following program defining $\texttt{half}_{\texttt{lst}}(A, B)$, which computes the last half of a list[3], illustrates why this may be problematic:

$$
\begin{array}{|l}
\hline
\texttt{half}_{\texttt{lst}}(A,B)\texttt{:-}\ \texttt{reverse}(A,C), \\
\qquad\qquad\qquad\qquad \texttt{case}_{\texttt{list}}(p_{[\,]}, p_{[H|T]}, C, B). \\
\qquad\quad \texttt{p}_{[\,]}(A)\texttt{:-}\ \texttt{empty}(A). \\
\texttt{p}_{[H|T]}(A,B,C)\texttt{:-}\ \texttt{empty}(B), \texttt{empty}(C). \\
\texttt{p}_{[H|T]}(A,B,C)\texttt{:-}\ \texttt{front}(B,D)^{4}, \\
\qquad\qquad\qquad\qquad \underline{\texttt{case}_{\texttt{list}}(p_{[\,]}, p_{[H|T]}, D, E)}, \\
\qquad\qquad\qquad\qquad \texttt{append}(E,A,C). \\
\hline
\end{array}
$$

The HO predicate $\texttt{case}_{\texttt{list}}(p_{[\,]}, p_{[H|T]}, A, B)$ calls $p_{[\,]}$ if A is empty and $p_{[H|T]}$ otherwise. Our definition of $half_{\texttt{lst}}(A, B)$ cannot be found using the standard search procedure as every occurrence of $\texttt{case}_{\texttt{list}}$ results in a call to the meta-interpreter's proof procedure. The underlined call to $\texttt{case}_{\texttt{list}}$ results in PI for $\texttt{p}_{[H|T]}$ *ad infinitum*. Similarly, the initial goal cannot be substituted unless it's explicitly specified.

As with $half_{\texttt{lst}}(A, B)$, The following program defining $\texttt{issubtree}(A, B)$, which computes whether B is a subtree of A, requires recursively calling `issubtree` through `any`.

$$
\begin{array}{|l}
\hline
\texttt{issubtree}(A,B)\texttt{:-}\ A = B. \\
\texttt{issubtree}(A,B)\texttt{:-}\ \texttt{children}(A,C), \texttt{any}(\texttt{cond}, C, B). \\
\qquad\quad \texttt{cond}(A,B)\texttt{:-}\ \underline{\texttt{issubtree}(A,B)}. \\
\hline
\end{array}
$$

This can be resolved using *metatypes* (see Section 6.5), but this is non-standard, results in a strong language bias, and does not always work. *Hopper* successfully learns these predicates without any significant drawbacks.

Negation of invented predicates (HO arguments of BK_{in} definitions), to the best of our knowledge, is not fully supported by $Metagol_{\text{HO}}$ (See Section 4.2 of [CMM20]). *Hopper* has similar issues which are discussed in Section 6.4.4.

Higher-order *HEXMIL*

HEXMIL is an ASP encoding of Meta-interpretive Learning [KEI18]. Given that ASP can be quite restrictive, *HEXMIL* exploits the HEX formalism for encoding MiL. HEX allows the ASP solver to interface with external resources [ERS16]. *HEXMIL* is restricted to *forward-chained metarules*:

Definition 6.2. *Forward-chained* metarules are of the form: $P(A, B)$:- $Q_1(A, C_1), Q_2(C_1, C_2), \cdots, Q_n(C_{n-1}, B), R_1(D_1), \cdots, R_m(D_m)$ where $D_i \in \{A, C_1, \cdots, C_{n-1}, B\}$.

Thus, only Dyadic learning task may be handled. Furthermore, many useful metarules are not of this form, i.e. $P(A, B)$:- $Q(A, B), R(A, B)$. $HEXMIL_{\text{HO}}$, incorporates HO

[3] $\texttt{half}_{\texttt{lst}}([1,2],[2])$, $\texttt{half}_{\texttt{lst}}([1,2,3],[3])$, $\texttt{half}_{\texttt{fst}}([1,2,3],[1,2])$.

[4] $\texttt{front}(A, B)$:- $\texttt{reverse}(A, C), \texttt{tail}(C, D), \texttt{reverse}(D, B)$.

definitions into the forward-chained structure of Definition 6.2. For details concerning the encoding see Section 4.4 of [CMM20]. The authors of [CMM20] illustrated $HEXMIL_{\mathrm{HO}}$'s poor performance on list manipulation tasks and its limitations make application to tasks of interest difficult. Thus, we focus on $Metagol_{\mathrm{HO}}$ in Section 6.5.

6.3.3 Popper: Learning From Failures (LFF)

The LFF paradigm together with *Popper* provides a novel approach to inductive logic programming, based on counterexample guided inductive synthesis (CEGIS) [SL08]. Both LFF and the system implementing it were introduced by A. Cropper and R. Morel [CM21a]. As input, *Popper* takes a set of predicate declarations PD, sets of positive E^+ and negative E^- examples, and background knowledge BK, the typical setting for *learning from entailment* ILP [Rae08].

During the **generate phase**, candidate programs are chosen from the viable hypothesis space, i.e. the space of programs that have yet to be ruled out by generated constraints. The chosen program is then tested (**test phase**) against E^+ and E^-. If only some of E^+ and/or some of E^- is entailed by the candidate hypothesis, Popper builds constraints (**constrain phase**) which further restrict the viable hypothesis space searched during the **generate phase**. When a candidate program only entails E^+, *Popper* terminates.

Popper searches through a finite hypothesis space, parameterized by features of the language bias (i.e. number of body predicates, variables, etc.). Importantly, if an optimal solution is present in this parameterized hypothesis space, Popper will find it (Theorem 1l [CM21a]). Optimal is defined as the solution containing the fewest literals [CM21a].

An essential aspect of this *generate, test, constrain* loop is the choice of constraints. Depending on how a candidate program performs in the **test phase**, Popper introduces constraints pruning specializations and/or generalizations of the candidate program. Specialization/generalization is defined via Θ-subsumption [Plo70, Rey70]. *Popper* may also introduce **elimination** Constraints pruning *separable*[5] sets of clauses. Details concerning the benefits of this approach are presented in [CM21a]. Essentially, *Popper* refines the hypothesis space, not the program [Sri01, Mug95, QCJ93].

In addition to constraints introduced during the search, like the majority of ILP systems, *Popper* incorporates a form of *language bias* [NCWSC97], that is predefined syntactic and/or semantic restrictions of the hypothesis space. *Popper* minimally requires *predicate declarations*, i.e. whether a predicate can be used in the head or body of a clause, and with what arities the predicate may appear. Popper accepts *mode declaration*-like hypothesis constraints [Mug95] which declare, for each argument of a given predicate, the type, and direction. Additional hypothesis constraints can be formulated as ASP programs (mentioned in Section 6.3.1).

Popper implements the *generate, test, constrain* loop using a *multi-shot solving* framework [GKKS19] and an encoding of both definite logic programs and constraints within the *ASP* [Lif19] paradigm. The language bias together with the generated constraints is encoded as an ASP program. The ASP solver is run on this program and the resulting

[5]No head literal of a clause in the set occurs as a body literal of a clause in the set.

model (if one exists) is an encoding of a candidate program.

6.4 Theoretical Framework

We provide a brief overview of logic programming. Our exposition is far from comprehensive. We refer the interested reader to a more detailed source [Llo87].

6.4.1 Preliminaries

Let $\mathcal{P}$ be a countable set of *predicate symbols* (denoted by $p, q, r, p_1, q_1, \cdots$), $\mathcal{V}_f$ be a countable set of *first-order (FO) variables* (denoted by $A, B, C, \cdots$) , and $\mathcal{V}_h$ be a countable set of *HO variables* (denoted by $P, Q, R, \cdots$). Let $\mathcal{T}$ denote the set of FO terms constructed from a finite set of function symbols and $\mathcal{V}_f$ (denoted by $s, t, s_1, t_1, \cdots$).

An *atom* is of the form $p(T_1, \cdots, T_m, t_1, \cdots, t_n)$. Let us denote this atom by a, then $sy(a) = p$ is the *symbol of the atom*, $ag_h(a) = \{T_1, \cdots, T_m\}$ are its *HO-arguments*, and $ag_f(a) = \{t_1, \cdots, t_n\}$ are its *FO-arguments*. When $ag_h(a) = \varnothing$ and $sy(a) \in \mathcal{P}$ we refer to a as *FO*, when $ag_h(a) \subset \mathcal{P}$ and $sy(a) \in \mathcal{P}$ we refer to a as *HO-ground*, otherwise it is *HO*. A *literal* is either an atom or its negation. A literal is *HO* if the atom it contains is *HO*. [6]

A *clause* is a set of literals. A *Horn* clause contains at most one positive literal while a definite clause must have exactly one positive literal. The atom of the positive literal of a definite clause c is referred to as the *head* of c (denoted by $hd(c)$), while the set of atoms of negated literals is referred to as the *body* (denoted by $bd(c)$). A *function-free definite (f.f.d) clause* only contains variables as FO arguments. We refer to a finite set of clauses as a *theory*. A theory is considered *FO* if all atoms are *FO*.

Replacing variables $P_1, \cdots, P_n, A_1, \cdots, A_m$ by predicate symbols $p_1, \cdots, p_n$ and terms $t_1, \cdots, t_m$ is a *substitution* (denoted by $\theta, \sigma, \cdots$) $\{P_1 \mapsto p_1, \cdots, P_n \mapsto p_n, A_1 \mapsto t_1, \cdots, A_m \mapsto t_m\}$. A substitution θ *unifies* two atoms when $a\theta = b\theta$.

6.4.2 Interpretable Theories and Groundings

Our hypothesis space consists of a particular type of theory which we refer to as *interpretable*. From these theories, one can derive so-called, *principle programs*, FO clausal theories encoding the relationship between certain literals and clauses and a set of higher-order definitions. *Hopper* generates and tests *principal programs*. This encoding preserves the soundness of the pruning mechanism presented in [CM21a]. Intuitively, the soundness follows from each principal program encoding a unique HO program. A consequence of this approach is that each HO program may be encoded by multiple *principal programs*, some of which may not be in a subsumption relation to each other, i.e. not *mutually prunable*. This results in a larger, though more expressive hypothesis space.

Definition 6.3. A clause c is *proper*[7] if $ag_h(hd(c))$ are pairwise distinct, $ag_h(hd(c)) \subset \mathcal{V}_h$, and $\forall a \in bd(c)$,

[6] $sy(l)$, $ag_h(l)$, and $ag_f(l)$ apply to literals with similar affect.
[7] Similar to definitional HO of W. Wadge [Wad91].

a) if $sy(a) \in \mathcal{V}_h$, then $sy(a) \in ag_h(hd(c))$, and

b) if $p \in ag_h(a)$ and $p \in \mathcal{V}_h$, then $p \in ag_h(hd(c))$.

A finite set of proper clauses d with the same head (denoted $hd(d)$) is referred to as a *HO definition*. A set of distinct HO definitions is a *library*. Let $\mathcal{P}_{PI} \subset \mathcal{P}$ be a set of predicate symbols reserved for invented predicates.

Definition 6.4. A f.f.d theory $\mathfrak{T}$ is *interpretable* if $\forall c \in \mathfrak{T}$, $ag_h(hd(c)) = \varnothing$ and $\forall l \in bd(c)$, l is higher-order ground,

a) if $ag_h(l) \neq \varnothing$, then $\forall c' \in \mathfrak{T}$, $sy(hd(c')) \neq sy(l)$, and

b) $\forall p \in ag_h(l), \exists c' \in \mathfrak{T}$, s.t. $sy(hd(c')) = p \in \mathcal{P}_{PI}$.

Atoms s.t. $ag_h(l) \neq \varnothing$ are *external*. The set of external atoms of an interpretable theory $\mathfrak{T}$ is denoted by $ex(\mathfrak{T})$.

Let $S_{PI}(\mathfrak{T}) = \{p_i \mid p_i \in ag_h(a) \wedge a \in ex(\mathfrak{T})\}$, the set of predicates which need to be invented. During the **generate phase** we enforce invention of $S_{PI}(\mathfrak{T})$ by pruning programs which contain external literals, but do not contain clauses for their arguments. We discuss this in more detail in Section 6.4.3.

Otherwise, interpretable theories do not require significant adaption of *Popper*'s *generate, test, constrain loop* [CM21a]. The HO arguments of external literals are ignored by the ASP solver, which searches for so-called *principal programs* (an FO representation of interpretable theories).

Example 6.5. Consider `reverse`, $\texttt{half}_{\texttt{lst}}$, and `issubtree` of Section 6.3.1 & 6.3.2. Each is an interpretable theory. The sets of external literals of these theories are $\{\texttt{fold}(p, C, A, B)\}$, $\{\texttt{case}_{\texttt{list}}(p_{[\,]}, p_{[H|T]}, C, B), \texttt{case}_{\texttt{list}}(p_{[\,]}, p_{[H|T]}, D, E)\}$, and $\{\texttt{any}(\texttt{cond}, C, B)\}$, respectively.

Definition 6.6. Let L be a library, and $\mathfrak{T}$ an interpretable theory. $\mathfrak{T}$ is *L-compatible* if $\forall l \in ex(\mathfrak{T}), \exists! d \in L.$ s.t. $hd(d)\sigma = l$ for some substitution σ. Let $df(L, l) = d$ and $\theta(L, l) = \sigma$.

Example 6.7. The program in Section 6.3.1 is L-compatible with the following library $L =$

$$\begin{array}{|l|}\hline \texttt{fold}(P, A, B, C) \texttt{:-}\ \texttt{empty}(B), C = A. \\ \texttt{fold}(P, A, B, C) \texttt{:-}\ \texttt{head}(B, H), \texttt{P}(A, H, D), \\ \qquad\qquad\qquad\qquad \texttt{tail}(B, T), \texttt{fold}(P, D, T, C). \\ \hline \end{array}$$

Let $l = fold(p, C, A, B)$: $df(L, l) = \texttt{fold}(P, A, B, C)$ and $\theta(L, l) = \{P \mapsto p, A \mapsto C, B \mapsto A, C \mapsto B\}$.

An L-compatible theory $\mathfrak{T}$ can be *L-grounded*. This requires replacing external literals of $\mathfrak{T}$ by FO literals, i.e. removal of all HO arguments and replacing the predicate symbol of the external literals with fresh predicate symbols, resulting in $\mathfrak{T}^*$, and adding clauses

that associate the FO literals l with the appropriate $d \in L$ and argument instantiations. different occurrences of external literals with the same symbol and same HO arguments result in FO literals with the same predicate symbol. The principal program contains all clauses derived from $\mathfrak{T}$, i.e. $\mathfrak{T}^*$ (See Example 6.8).

Example 6.8. Using the library of Example 6.7 and a modified version of the program from Section 6.3.1 (p is replaced by $\texttt{fold}_{p\_a}$ for clarity purposes), we get the following L-grounding:

$$
\begin{aligned}
&\texttt{reverse}(A,B)\text{:- }\texttt{empty}(C),\ \texttt{fold}_a(C,A,B).\\
&\texttt{fold}_{p\_a}(A,B,C)\text{:- }\texttt{head}(C,B),\ \texttt{tail}(C,A).\\
&\texttt{fold}_a(A,B,C)\text{:- }\texttt{fold}(\texttt{fold}_{p\_a},A,B,C).\\
&\texttt{fold}(P,A,B,C)\text{:- }\texttt{empty}(B),A=C.\\
&\texttt{fold}(P,A,B,C)\text{:- }\texttt{head}(B,H),P(H,D),\\
&\qquad\qquad\qquad\qquad\texttt{tail}(B,T),\texttt{fold}(P,D,T,C).
\end{aligned}
$$

$\texttt{fold}_a(C,A,B)$ replaces $\texttt{fold}(\texttt{fold}_{p\_a},C,A,B)$. The first two clauses form the principal program.

If an *L-compatible* theory contains multiple external literals whose symbol is fold, i.e. $\texttt{fold}(\texttt{fold}_{p\_a},C,A,B)$ and $\texttt{fold}(\texttt{fold}_{p\_a},D,E,R)$, both are renamed to $\texttt{fold}_a$. However, if the higher-order arguments differ, i.e. $\texttt{fold}_{p\_a}$, and $\texttt{fold}_{p\_b}$, then they are renamed to $\texttt{fold}_a$ and $\texttt{fold}_b$, and an additional clause $\texttt{fold}_b(A,B,C)$:- $\texttt{fold}(\texttt{fold}_{p\_b},A,B,C)$ would be added to the L-grounding. When a definition takes more than one HO argument and arguments of instances partially overlap, duplicating clauses may be required during the construction of the L-grounding. Soundness of the pruning mechanism is preserved because the FO literals uniquely depend on the arguments fed to HO definitions.

Note, the system requires the user to provide higher-order definitions, similar to *Metagol*$_{\text{HO}}$. Additionally, these HO definitions may be of the form $ho(P,Q,x,y)$:- $P(Q,x,y)$, essentially a *higher-order definition template*. While allowed by the formalism, we have not thoroughly investigated such constructions. This amounts to the invention of HO definitions.

6.4.3 Interpretable Theories and Constraints

The constraints of Section 6.3.3 are based on *Θ-subsumption*:

Definition 6.9 (Θ-subsumption)**.** An FO theory T_1 subsumes an FO theory T_2, denoted by $T_1 \leq_\theta T_2$ iff, $\forall c_2 \in T_2 \exists c_1 \in T_1$ s.t. $c_1 \leq_\theta c_2$, where $c_1 \leq_\theta c_2$ iff, $\exists\theta$ s.t. $c_1\theta \subseteq c_2$.

Importantly, the following property holds:

Proposition 6.10. *if* $T_1 \leq_\theta T_2$*, then* $T_1 \models T_2$

The pruning ability of *Popper*'s Generalization and specialization constraints follows from Proposition 6.10.

Definition 6.11. An FO theory T_1 is a *generalization* (*specialization*) of an FO theory T_2 iff $T_1 \leq_\theta T_2$ ($T_2 \leq_\theta T_1$).

Given a library L and a space of L-compatible theories, we can compare L-groundings using Θ-subsumption and prune generalizations (specializations), based on the **Test phase**.

Groundings and Elimination Constraints

During the **generate phase**, elimination constraints prune separable programs (See Footnote 5). While L-groundings are non-separable, and thus avoid pruning in the presence of elimination constraints, it is inefficient to query the ASP solver for L-groundings. The ASP solver would have to know the library and how to include definitions. Furthermore, the library must be written in an ASP-friendly form [CM21a]. Instead, we query the ASP solver for the *principal program*. The definitions from the library L are treated as BK. Consider Example 6.8, during the **generate phase** the ASP solver may return an encoding of the following clauses:

$$\boxed{\begin{array}{l}\texttt{reverse}(A,B)\texttt{:- empty}(C),\ \texttt{fold}(C,A,B).\\ \quad\texttt{p}(A,B,C)\texttt{:- head}(C,B),\ \texttt{tail}(C,A).\end{array}}$$

During the **test phase** the rest of the L-grounding is re-introduced. While this eliminates inefficiencies, the above program is now separable and may be pruned. To efficiently implement HO synthesis we relaxed the elimination constraint in the presence of a library. Instead, we introduce so-called *call graph constraints* defining the relationship between HO literals and auxiliary clauses. This is similar to the *dependency graph* introduced in [CM21b].

6.4.4 Negation, Generalization, and Specialization

Negation (under classical semantics) of HO literals can interfere with *Popper* constraints. Consider the ILP task and candidate programs:

$$E^+ : f(b).\quad f(c). \qquad\qquad \mathbf{E}^- : f(a).$$

$$BK : \left\{\begin{array}{ll} p(a). & p(b).\\ q(a). & q(c).\end{array}\right\} \qquad\qquad \mathbf{HO} : N(P,A)\texttt{:- } \neg\ P(A).$$

$\underline{prog_s}$

$$\boxed{\begin{array}{l}\texttt{f}(A)\texttt{:- N}(p_1,A).\\ \texttt{p}_1(A)\texttt{:- p}(A),\ \texttt{q}(A).\end{array}}$$

$\underline{prog_f}$

$$\boxed{\begin{array}{l}\texttt{f}(A)\texttt{:- N}(p_1,A).\\ \texttt{p}_1(A)\texttt{:- p}(A).\end{array}}$$

The optimal solution is $prog_s$, $prog_f$ is an *incorrect* hypothesis which *Hopper* can generate prior to $prog_s$, and $prog_f \leq_\theta prog_s$. Note, $prog_f \models \neg f(b) \wedge \neg f(a) \wedge f(c)$, it does not entail all of E^+. We should generalize $prog_f$ to find a solution, i.e. add literals to p_1. The introduced constraints [CM21a] prune programs extending p_1, i.e. $prog_s$. Similar

holds for specializations. Consider the ILP task and candidate programs:

$E^+ : f(a).\quad f(b).$ $\qquad$ $\mathbf{E}^- : f(c).\quad f(d).$

$BK : \left\{\ p(d).\quad q(c).\ \right\}$ $\qquad$ $\mathbf{HO} : N(P,X)\text{:-}\ \neg\ P(X).$

$\underline{prog_s}$ $\qquad$ $\underline{prog_f}$

```
f(A):- N(p1,A).
p1(A):- p(A).
p1(A):- q(A).
```

```
f(A):- N(p1,A).
p1(A):- p(A).
```

The optimal solution is $prog_s$, $prog_f$ is an *incorrect* hypothesis which *Hopper* can generate prior to $prog_s$, and $prog_s \leq_\theta prog_f$. Note, $prog_f \models f(a) \wedge f(b) \wedge f(c)$, it entails some of E^-. We should specialize p_1 to find a solution, i.e. add clauses to $prog_f$. The introduced constraints [CM21a] prune programs that add clauses, i.e. $prog_s$.

Handling negation of invented predicates is feasible but non-trivial as it would require significant changes to the constraint construction procedure. We leave it to future work.

6.5 Experiments

A possible, albeit very weak, program synthesizer is an enumeration procedure that orders all possible programs constructible from the *BK* by size, testing each until a solution is found. In [CM21a], this procedure was referred to as *Enumerate*. *Popper* extends *Enumerate* by pruning the hypothesis space based on previously tested programs.

The pruning mechanism will never prune the shortest solution. Thus, the important question when evaluating *Popper*, and *Hopper*, is not if *Popper* will find a solution, nor is it a high-quality solution, but rather how long it takes *Popper* to find the solution. An extensive suite of experiments was presented in [CM21a], illustrating that *Popper* outperforms *Enumerate* and existing ILP systems.

One way to improve the performance of *LFF*-based ILP systems, like *Popper*, is to introduce techniques that shorten or simplify the solution. The authors of [CMM20], in addition to introducing $Metagol_{\text{HO}}$ and $HEXMIL_{\text{HO}}$, provided a comprehensive suite of experiments illustrating that the addition of HO predicates can improve accuracy and, most importantly, reduce learning time. Reduction in learning times results from a reduction in the complexity/size of the solutions.

The experiments in [CM21a] thoroughly cover scalability issues and learning performance on simple list transformation tasks, but do not cover performance on complex tasks with large solutions. The experiments presented in [CMM20] illustrate performance gains when a HO library is used to solve many simple tasks and how the addition of HO predicates allows the synthesis of relatively complex predicates such as `dropLast`. When the solution is large *Popper*'s performance degrades significantly. When the solution requires complex interaction between predicates and clauses it becomes exceedingly

difficult to find a set of metarules for $Metagol_{\text{HO}}$without being overly descriptive or suffering from long learning times.

Our experiments illustrate that the combination of *Popper* and HO predicates [CMM20] significantly improves *Popper*'s performance at learning complex programs. Similar to [CM21a], we use predicate declarations, i.e. `body_pred(head,2)`, type declarations, i.e. `type(head,(list,element))`, direction declarations, i.e. `direction(head,(in,out))`, and the parameters required by *Popper*'s search mechanism, `max_var`, `max_body`, and `max_clauses`.

We reevaluated 7 of the tasks presented in [CM21a] and 2 presented in [CMM20]. Additionally, we added 8 list manipulation tasks, 3 tree manipulation tasks, and 2 arithmetic tasks (separated by type in Table 6.1). Our additional tasks are significantly harder than the tasks evaluated in previous work.

For each task, we guarantee that the optimal solution is present in the hypothesis space and record how long *Popper* and *Hopper* take to find it. We ran *Popper* using optimal settings and minimal *BK*. In some cases, the tasks cannot be solved by *Popper* without *Predicate Invention* (See Column **PI?** of Table 6.1), i.e. a explanatory hypothesis which is both accurate and precise requires auxiliary concepts.

We ran *Hopper* in two modes, Column ***Hopper*** concerns running *Hopper* with the same settings and a superset of the *BK* used by *Popper* (minus constructions used to force invention), while Column ***Hopper* (Opt)** concerns running *Hopper* with optimal settings and minimal *BK*. For *Popper* and *Hopper*, settings such as `max_var` significantly impact performance. Both systems search for the shortest program (by literal count) respecting the current constraints. Note, that hypothesizing a program with an additional clause w.r.t the previously generated programs requires increasing the literal count by at least two. Thus, the current search procedure avoids such hypotheses until all shorter programs have been pruned or tested. The parameters `max_var` and `max_body` have a significant impact on the size of the single clause hypothesis space. Given that the use of HO definitions always requires auxiliary clauses, using large values, for the above-mentioned parameter, will hinder their use. This is why *Hopper* performs significantly better post-optimization. Using a comparably large *BK* incurs an insignificant performance impact compared to using unintuitively large parameter settings (see Proposition 1, page 14 [CM21a]).

The predicates used for a particular task are listed in Column *HO-Predicate* of Table 6.1. *Popper* and *Hopper* timed out (300 seconds elapsed) when large clauses or many variables are required. Timing out means the optimal solution was not found in 300 seconds. Given that we know the solution to each task, Column **#Literals** provides the size of the known solution, not the size of the non-optimal solution found by the system in case of timeout.

Concerning the Optimizations, these runs of *Hopper* closely emulate how such a system would be used as it is pragmatic to search assuming smaller clauses and fewer variables are sufficient and expand the space as needed. *Popper*'s and *Hopper*'s performance degrades by just assuming the solution may be complex. For *findDup*, *Hopper* found the FO solution.

Overall, this *optimization* issue raises a question concerning the search mechanism

currently used by both *Popper* and *Hopper*. While HO solutions are typically shorter than the corresponding FO solution, this brevity comes at the cost of a complex program structure. This trade-off is not considered by the current implementations of the LFF paradigm. Investigating alternative search mechanisms and optimality conditions (other than literal count) is planned future work.

We attempted to solve each task using $Metagol_{\text{HO}}$. Successful learning using $Metagol_{\text{HO}}$ is highly dependent on the choice of the metarules. To simplify matters, we chose metarules that mimic the clauses found in the solution. In some cases, this requires explicitly limiting how certain variables are instantiated by adding declarations, i.e. `metagol:type(Q,2,head_pred)`, to the body of a metarule (denoted by *metatype* in Table 6.1). This amounts to significant human guidance, and thus, both simplifies learning and what we can say comparatively about the system. Hence we limit ourselves to indicating success or failure only.

Under these experimental settings, every successful task was solved faster by $Metagol_{\text{HO}}$than *Hopper* with optimal settings. Relaxing restrictions on the metarule set introduces a new variable into the experiments. Choosing a set of metarules that is general and covers every task will likely result in the failure of the majority of tasks. Some tasks require splitting rules such as $P(A, B)$:- $Q(A, B), R(A, B)$ which significantly increase the size of the hypothesis space. Choosing metarules per task, but without optimizing for success, leaves the question, *which metarules are appropriate/acceptable for the given task?* While this is an interesting question [CT20], the existence problem of a set of *general* metarules over which $Metagol_{\text{HO}}$'s performance is comparable to, or even better than, *Hopper*'s only strengthens our argument concerning the chosen experimental setting as one will have to deduce/design this set.

6.6 Implementation

We implement our method by building on top of code provided by [CM21a]. The changes we applied include:

Processing HO predicates We allow user to declare some background knowledge predicates to be HO. Based on these declarations we generate ASP constraints discussed in subsection 6.4.2 and Prolog code that allows execution of programs generated with those constraints.

Generating context-passing versions of HO predicates Sometimes the HO argument predicates (referred to as $S_{PI}(\mathfrak{T})$ in subsection 6.4.2) require context that exists in the predicate that calls them, however is inaccessible to them in our framework. To make it accessible we support automating generation of *more contextual* versions of HO predicates. These predicates have higher arity and take more FO arguments. These arguments are only used in HO calls, and are simply passed as arguments to HO predicate calls. In [CMM20] this process is referred to as „currying“ (though it is somewhat different to what *currying* is usually considered to be).

Example 6.12. From a HO `map` predicate

$$\begin{array}{l}\texttt{map}(P,[\,],[\,]).\\ \texttt{map}(P,[H_1|T_1],[H_2|T_2])\text{:-}\ P(H_1,H_2),\texttt{map}(P,T_1,T_2).\end{array}$$

we automatically generate a more contextual version

$$\begin{array}{l}\texttt{map}(P,[\,],[\,],V).\\ \texttt{map}(P,[H_1|T_1],[H_2|T_2],V)\text{:-}\ P(H_1,H_2,V),\\ \qquad\qquad\qquad\qquad\qquad\texttt{map}(P,T_1,T_2,V).\end{array}$$

which allows for construction of a program that adds a number to every element of a list using `map`

$$\begin{array}{l}\texttt{f}(A,B,C)\text{:-}\ \texttt{map}(\texttt{p}_1,B,C,A).\\ \texttt{p}_1(A,B,C)\text{:-}\ \texttt{add}(A,C,B).\end{array}$$

Forcing all generated code to be used Since ASP can now generate many different predicates, some of them might not even be called in the main predicate. To avoid such useless code we make ASP keep track of a *call graph* – which predicates call which other predicates, and add a constraint that forces every defined predicate to be called (possibly indirectly) by the main predicate. This not only removes many variations of effectively the same program but also significantly prunes the hypothesis space, pruning programs ignored by other constraints (explained in sec. 6.4.3).

Changes to separability and recursion We add a few small changes to solve the problems that appear when generating multiple predicates. We make sure that clauses that call HO predicates (and thus different predicates from the program) are not considered *separable*. We also change how recursion is handled – otherwise, recursion would allow all invented arguments to be called everywhere in the program, needlessly increasing search space.

6.7 Conclusion and Future Work

We extended the LFF-based ILP system *Popper* to effectively use user-provided HO definitions during learning. Our experiments show *Hopper* (when optimized) is capable of outperforming *Popper* on most tasks, especially the harder tasks we introduced in this work (Section 6.5). *Hopper* requires minimal guidance compared to the top-performing MiL-based ILP system $Metagol_{\text{HO}}$. Our experiments test the theoretical possibility of $Metagol_{\text{HO}}$ finding a solution under significant guidance. However, given the sensitivity to metarule choice and the fact that many tasks have ternary and even 4-ary predicates, it is hard to properly compare these approaches.

We provide a theoretical framework encapsulating the accepted HO definitions, a fragment of the definitions monotone over subsumption and entailment, and discuss the limitations of this framework. A detailed account is provided in Section 6.4. The

main limitation of this framework concerns HO-negation which we leave to future work. Our framework also allows for the invention of HO predicates during learning through constructions of the form $ho(P, Q, x, y)$:- $P(Q, x, y)$. We can verify that *Hopper* can, in principle, find the solution, but we have not successfully invented an HO predicate during learning. We plan for further investigation of this problem.

As briefly mentioned, *Hopper* was tested twice during experimentation due to the significant impact system parameters have on its performance. This can be seen as an artifact of the current implementation of LFF which is bias towards programs with fewer, but longer, clauses rather than programs with many short clauses. An alternative implementation of LFF taking this bias into account, together with other bias, is left to future investigation.

Acknowledgements

We would like to thank Rolf Morel and Andrew Cropper for their thorough commentary which helped us greatly improve a preliminary version of this paper. Supported by the ERC starting grant no. 714034 SMART, the Math$_{LP}$ project (LIT-2019-7-YOU-213) of the Linz Institute of Technology and the state of Upper Austria, Cost action CA20111 EuroProofNet, and project CZ.02.2.69/0.0/0.0/18_053/0017594 of the Ministry of Education, Youth and Sports of the Czech Republic.

Task	*Popper* **(Opt)**	#Literals	PI?	*Hopper*	***Hopper* (Opt)**	#Literals	HO-Predicates	*Metagol*$_{\text{HO}}$	Metatypes?
Learning Programs by learning from Failures [CM21a]									
dropK	1.1s	7	no	0.5s	**0.1s**	4	iterate	no	no
allEven	0.2s	7	no	0.2s	**0.1s**	4	all	yes	no
findDup	0.25s	7	no	– –	**0.5s**	10	caseList	no	yes
length	0.1s	7	no	0.2s	**0.1s**	5	fold	yes	no
member	0.1s	5	no	0.2s	**0.1s**	4	any	yes	no
sorted	65.0s	9	no	46.3s	**0.4s**	6	fold	yes	no
reverse	11.2s	8	no	7.7s	**0.5s**	6	fold	yes	no
Learning Higher-Order Logic Programs [CMM20]									
dropLast	300.0s	10	no	300s	**2.9s**	6	map	yes	no
encryption	300.0s	12	no	300s	**1.2s**	7	map	yes	no
Additional Tasks									
repeatN	5.0s	7	no	0.6s	**0.1s**	5	iterate	yes	no
rotateN	300.0s	10	no	300s	**2.6s**	6	iterate	yes	no
allSeqN	300.0s	25	yes	300s	**5.0s**	9	iterate, map	yes	no
dropLastK	300.0s	17	yes	300s	**37.7s**	11	map	no	no
firstHalf	300.0s	14	yes	300s	**0.2s**	9	iterateStep	yes	no
lastHalf	300.0s	12	no	300s	**155.2s**	12	caseList	no	yes
of1And2	300.0s	13	no	300s	**6.9s**	13	try	no	no
isPalindrome	300.0s	11	no	157s	**2.4s**	9	condlist	no	yes
depth	300.0s	14	yes	300s	**10.1s**	8	fold	yes	yes
isBranch	300.0s	17	yes	300s	**25.9s**	12	caseTree, any	no	yes
isSubTree	2.9s	11	yes	1.0s	**0.9s**	7	any	yes	yes
addN	300.0s	15	yes	300s	**1.4s**	9	map, caseInt	yes	no
mulFromSuc	300.0s	19	yes	300s	**1.2s**	7	iterate	yes	no

Table 6.1: We ran *Popper*, *Hopper*, *optimized Hopper* , and *Metagol*$_{\text{HO}}$ on a single core with a timeout of 300 second. Times denote the average of 5 runs. Evaluation time for *Popper* and *Hopper* was set to a thousandth of a second, sufficient time for all task involved.

Chapter 7

Conclusion

7.1 Summary

This thesis explored multiple approaches toward building an Artificial Intelligence system capable of abstract reasoning.

I have worked on Deep Neural Network architecture capable of working well with graphs, which allows representing well (almost) any abstract object. My proposed architecture solves the problem of differentiating between graphs that the Weisfeiler-Lehman isomorphism test [WL68] fails to distinguish, as well as allows taking into account far-away interactions. I have however not observed improvement in real-world graph classification benchmarks.

I have experimented with using Autoencoding to learn representations of mathematical formulae. I have tested two ways of computing loss for this setup and found the recursive way to generate much better results.

I have proposed a way to learn mathematical reasoning from scratch, being given only a set of mathematical axioms. This approach is based on the AlphaZero [SHS+18] algorithm, which can learn to play an arbitrary two-player board game (like Chess or Go). We define a two-player theorem construction game, where one player (the *adversary*) constructs a provable theorem and the other (the *prover*) attempts to prove it. We have shown this approach to be better than learning from uniformly randomly generated theorems, but achieved worse results than initially hoped for.

I have also worked on Inductive Logic Programming (a machine learning method that explicitly uses abstract reasoning). Together with co-authors, we have improved a recently introduced *Popper* [CM21a] system by allowing it to use higher-order predicates.

I worked on improving an older ILP system, *δILP* [EG18], that works by gradient descent – a method used in Deep Learning. This potentially allows it to be used in tandem with Deep Learning methods. My improvement consisted of allowing the system to use way more invented predicates, therefore performing gradient descent in much higher-dimensional space.

7.2 Future work

Since I have implemented my AlphaZero-based experiments (see chapter 5), many replications and investigations [Wu19, GAT+20] of the original work have been published, providing hints about how to achieve good results even without the computational

resources required originally. These include things like randomly varying the number of playouts in MCTS search and changing the way the playouts are distributed. Applying these to my work is a promising new direction of research.

There are also several decisions I would have made differently with my current knowledge: for example, the players would construct proofs of *rules* rather than *terms*. This would also allow for easy reuse of already proven theorems in new proofs, another new potential feature.

There is also the problem of minimum required unification: a fact, that any constructed theorem would only be unified as little as necessary, leaking valuable hints about the proof. Giving the adversary a way around this problem is something to investigate.

Another way of using this work would be to augment algorithms learning from a dataset of human generated theorems (for example [KUMO18]). Such systems can get stuck on „gaps" in curriculum. The adversarial generation of additional curriculum could be useful in overcoming this problem.

My work on higher-order ILP (chapter 6) can also be taken further in multiple ways. One such way is to allow *Popper* to not only use but also synthesize higher-order predicates – something we have not done thus far. Since the synthesis of higher-order predicates is only useful when the synthesized predicate is used multiple times in a program, this would probably require solving multiple problems simultaneously. That would ensure that the shortest solution (to all problems at once) would be making use of synthetic HO predicates. I would unfortunately also make finding the solution harder.

A different approach to synthesizing higher-order predicates is compressing a set of programs by introducing a new predicate – a higher-order one. It being HO would make it more likely that some common part can be found, but would make the search less straightforward.

Likewise, there are many ways to continue my work on differentiable ILP. The first, most obvious is to try again training a hybrid DNN-ILP system end-to-end. Such a system was described in the original work [EG18], but its parts were trained separately – presumably because training them together did not work. As our approach produces better results in most cases, it might make such a system plausible. Hopefully, this system would benefit from both DNN's ability to learn complex patterns and ILP's ability to generalize well to unseen data.

There is also a possibility of using auxiliary losses when learning programs. Such loss could achieve things like minimizing the size of the program or avoiding fuzzy solutions.

The gradient computation itself might also be optimized, using some kind of stochastic estimation similar to the one used in the REINFORCE algorithm [Wil92].

Finally, the general principle of dividing a program into small parts (each one possibly referencing other parts) and learning it by gradient descent can be used outside of Logic Programming – for example synthesizing a functional or imperative program.

The functional approach is of particular interest since neural network architectures can be easily expressed as functional code. The functional program we learn could therefore

be the architecture of the neural network we would be training at the same time. Thus, we would be training and designing the structure of a neural network at the same time. Even more interestingly, one could create a dynamically routed neural network, able to reuse its parts during inference. Additionally, if we allowed for recursion, this system could allow for truly Turing-complete neural networks, capable of performing an arbitrary amount of computation before generating an answer.

Bibliography

[ACE+16] Alexander A. Alemi, François Chollet, Niklas Eén, Geoffrey Irving, Christian Szegedy, and Josef Urban. DeepMath - deep sequence models for premise selection. In Daniel D. Lee, Masashi Sugiyama, Ulrike V. Luxburg, Isabelle Guyon, and Roman Garnett, editors, *Advances in Neural Information Processing Systems 29: Annual Conference on Neural Information Processing Systems 2016, December 5-10, 2016, Barcelona, Spain*, pages 2235–2243, 2016.

[ACKS17] Miltiadis Allamanis, Pankajan Chanthirasegaran, Pushmeet Kohli, and Charles A. Sutton. Learning continuous semantic representations of symbolic expressions. In Doina Precup and Yee Whye Teh, editors, *Proceedings of the 34th International Conference on Machine Learning, ICML 2017*, volume 70 of *Proceedings of Machine Learning Research*, pages 80–88. PMLR, 2017.

[BBG+18] Grzegorz Bancerek, Czeslaw Bylinski, Adam Grabowski, Artur Kornilowicz, Roman Matuszewski, Adam Naumowicz, and Karol Pak. The role of the mizar mathematical library for interactive proof development in mizar. *J. Autom. Reason.*, 61(1-4):9–32, 2018.

[BC13] Yves Bertot and Pierre Castéran. *Interactive theorem proving and program development: Coq'Art: the calculus of inductive constructions*. Springer Science & Business Media, 2013.

[BCB14] Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. Neural machine translation by jointly learning to align and translate. *arXiv preprint arXiv:1409.0473*, 2014.

[BG19] Chad E. Brown and Thibault Gauthier. Self-learned formula synthesis in set theory, 2019.

[BGG97] Egon Börger, Erich Grädel, and Yuri Gurevich. *The classical decision problem*. Springer-Verlag Berlin Heidelberg, 1997.

[BGK+16] Jasmin Christian Blanchette, David Greenaway, Cezary Kaliszyk, Daniel Kühlwein, and Josef Urban. A learning-based fact selector for Isabelle/HOL. *J. Autom. Reasoning*, 57(3):219–244, 2016.

[BGLS92] Leo Bachmair, Harald Ganzinger, Christopher Lynch, and Wayne Snyder. Basic paramodulation and superposition. In Deepak Kapur, editor, *Automated Deduction - CADE-11, 11th International Conference on Automated*

Deduction, Saratoga Springs, NY, USA, June 15-18, 1992, Proceedings, volume 607 of *Lecture Notes in Computer Science*, pages 462–476. Springer, 1992.

[BKH16] Jimmy Lei Ba, Jamie Ryan Kiros, and Geoffrey E Hinton. Layer normalization. *arXiv preprint arXiv:1607.06450*, 2016.

[BKPU16] Jasmin C. Blanchette, Cezary Kaliszyk, Lawrence C. Paulson, and Josef Urban. Hammering towards QED. *J. Formalized Reasoning*, 9(1):101–148, 2016.

[Blo99] Hendrik Blockeel. Top-down induction of first order logical decision trees. *AI Communications*, 12(1–2):119–120, January 1999.

[BLRS19] Kshitij Bansal, Sarah M. Loos, Markus N. Rabe, and Christian Szegedy. Learning to reason in large theories without imitation. *ArXiv*, abs/1905.10501, 2019.

[BMR+20] 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. *CoRR*, abs/2005.14165, 2020.

[BN98] Franz Baader and Tobias Nipkow. *Term Rewriting and All That*. Cambridge University Press, New York, NY, USA, 1998.

[BR67] J. Barkley Rosser. Haskell b. curry and robert feys. combinatory logic. volume i. with two sections by william craig. studies in logic and the foundations of mathematics. north-holland publishing company, amsterdam1958, xvi 417 pp. *Journal of Symbolic Logic*, 32(2):267–268, 1967.

[BSR+19] Kshitij Bansal, Christian Szegedy, Markus N. Rabe, Sarah M. Loos, and Viktor Toman. Learning to reason in large theories without imitation, 2019.

[BvBW07] Patrick Blackburn, J. F. A. K. van Benthem, and Frank Wolter, editors. *Handbook of Modal Logic*, volume 3 of *Studies in logic and practical reasoning*. North-Holland, 2007.

[CAC+19] Maxwell Crouse, Ibrahim Abdelaziz, Cristina Cornelio, Veronika Thost, Lingfei Wu, Kenneth D. Forbus, and Achille Fokoue. Improving graph neural network representations of logical formulae with subgraph pooling. *ArXiv*, abs/1911.06904, 2019.

[CAR18] Saikat Chakraborty, Miltiadis Allamanis, and Baishakhi Ray. Tree2tree neural translation model for learning source code changes. *CoRR*, abs/1810.00314, 2018.

[CBS19] Ting Chen, Song Bian, and Yizhou Sun. Are powerful graph neural nets necessary? A dissection on graph classification. *CoRR*, abs/1905.04579, 2019.

[CCE+18] Peter Clark, Isaac Cowhey, Oren Etzioni, Tushar Khot, Ashish Sabharwal, Carissa Schoenick, and Oyvind Tafjord. Think you have solved question answering? try arc, the AI2 reasoning challenge. *CoRR*, abs/1803.05457, 2018.

[CD20] Andrew Cropper and Sebastijan Dumancic. Inductive logic programming at 30: a new introduction. *ArXiv*, abs/2008.07912, 2020.

[CDEM22] Andrew Cropper, Sebastijan Dumančić, Richard Evans, and Stephen H. Muggleton. Inductive logic programming at 30. *Machine Learning*, 111(1):147–172, Jan 2022.

[Cho19] Franccois Chollet. On the measure of intelligence. *ArXiv*, abs/1911.01547, 2019.

[CJRS13] Koen Claessen, Moa Johansson, Dan Rosén, and Nicholas Smallbone. Automating inductive proofs using theory exploration. In Maria Paola Bonacina, editor, *Automated Deduction – CADE-24*, pages 392–406, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.

[CJSU19] Karel Chvalovský, Jan Jakubuv, Martin Suda, and Josef Urban. ENIGMA-NG: efficient neural and gradient-boosted inference guidance for E. In Pascal Fontaine, editor, *Automated Deduction - CADE 27 - 27th International Conference on Automated Deduction, Natal, Brazil, August 27-30, 2019, Proceedings*, volume 11716 of *Lecture Notes in Computer Science*, pages 197–215. Springer, 2019.

[CK18] Łukasz Czajka and Cezary Kaliszyk. Hammer for Coq: Automation for dependent type theory. *J. Autom. Reasoning*, 61(1-4):423–453, 2018.

[CLB17] Zhengdao Chen, Xiang Li, and Joan Bruna. Supervised community detection with line graph neural networks. *arXiv preprint arXiv:1705.08415*, 2017.

[CM14] Andrew Cropper and Stephen H. Muggleton. Logical minimisation of metarules within meta-interpretive learning. In *Proceedings of the 24th International Conference on Inductive Logic Programming*, pages 62–75, Nancy, France, September 2014. Springer.

[CM16] Andrew Cropper and Stephen Muggleton. Metagol system. github.com/metagol/metagol, 2016.

[CM21a] Andrew Cropper and Rolf Morel. Learning programs by learning from failures. *Machine Learning*, 110(4):801–856, February 2021.

[CM21b] Andrew Cropper and Rolf Morel. Predicate invention by learning from failures. *CoRR*, abs/2104.14426, May 2021.

[CMM20] Andrew Cropper, Rolf Morel, and Stephen Muggleton. Learning higher-order logic programs. *Machine Learning*, 109(7):1289–1322, July 2020.

[Cro20] Andrew Cropper. Forgetting to learn logic programs. *Proceedings of the AAAI Conference on Artificial Intelligence*, 34(04):3676–3683, April 2020.

[CT20] Andrew Cropper and Sophie Tourret. Logical reduction of metarules. *Machine Learning*, 109(7):1323–1369, July 2020.

[Cul97] Joseph Culberson. Sokoban is pspace-complete. 1997.

[CVCB19] Zhengdao Chen, Soledad Villar, Lei Chen, and Joan Bruna. On the equivalence between graph isomorphism testing and function approximation with gnns. In *Advances in Neural Information Processing Systems*, pages 15868–15876, 2019.

[DCLT19] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. *ArXiv*, abs/1810.04805, 2019.

[DGMB19] Sebastijan Dumancic, Tias Guns, Wannes Meert, and Hendrik Blockeel. Learning relational representations with auto-encoding logic programs. In Sarit Kraus, editor, *Proceedings of the Twenty-Eighth International Joint Conference on Artificial Intelligence, IJCAI 2019, Macao, China, August 10-16, 2019*, pages 6081–6087. ijcai.org, 2019.

[DHS+19] Simon S Du, Kangcheng Hou, Russ R Salakhutdinov, Barnabas Poczos, Ruosong Wang, and Keyulu Xu. Graph neural tangent kernel: Fusing graph neural networks with graph kernels. In *Advances in Neural Information Processing Systems*, pages 5724–5734, 2019.

[DSSV19] George Dasoulas, Ludovic Dos Santos, Kevin Scaman, and Aladin Virmaux. Coloring graph neural networks for node disambiguation. *arXiv preprint arXiv:1912.06058*, 2019.

[EG18] Richard Evans and Edward Grefenstette. Learning explanatory rules from noisy data. *Journal of Artificial Intelligence Research*, 61:1–64, January 2018.

[ERS16] Thomas Eiter, Christoph Redl, and Peter Schüller. Problem solving using the HEX family. In *Proceedings of Computational Models of Rationality*, pages 150–174. College Publications, June 2016.

[FAA+21] 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, 2021.

[FB16] Michael Färber and Chad E. Brown. Internal guidance for satallax. In Nicola Olivetti and Ashish Tiwari, editors, *Automated Reasoning - 8th International Joint Conference, IJCAR 2016, Coimbra, Portugal, June 27 - July 2, 2016, Proceedings*, volume 9706 of *Lecture Notes in Computer Science*, pages 349–361. Springer, 2016.

[GAA+13] 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.

[GAT+20] Jean-Bastien Grill, Florent Altché, Yunhao Tang, Thomas Hubert, Michal Valko, Ioannis Antonoglou, and Rémi Munos. Monte-carlo tree search as regularized policy optimization. *CoRR*, abs/2007.12509, 2020.

[Gau20] Thibault Gauthier. Deep reinforcement learning for synthesizing functions in higher-order logic. In Elvira Albert and Laura Kovács, editors, *LPAR 2020: 23rd International Conference on Logic for Programming, Artificial Intelligence and Reasoning, Alicante, Spain, May 22-27, 2020*, volume 73 of *EPiC Series in Computing*, pages 230–248. EasyChair, 2020.

[Gir87] Jean-Yves Girard. Linear logic. *Theoretical computer science*, 50(1):1–101, 1987.

[GK96] Christoph Goller and Andreas Küchler. Learning task-dependent distributed representations by backpropagation through structure. *Proceedings of International Conference on Neural Networks (ICNN'96)*, 1:347–352 vol.1, 1996.

[GKKS19] Martin Gebser, Roland Kaminski, Benjamin Kaufmann, and Torsten Schaub. Multi-shot asp solving with clingo. *Theory and Practice of Logic Programming*, 19(1):27–82, July 2019.

[GKU16] Thibault Gauthier, Cezary Kaliszyk, and Josef Urban. Initial experiments with statistical conjecturing over large formal corpora. In Andrea Kohlhase, Paul Libbrecht, Bruce R. Miller, Adam Naumowicz, Walther Neuper, Pedro Quaresma, Frank Wm. Tompa, and Martin Suda, editors, *Joint Proceedings of the FM4M, MathUI, and ThEdu Workshops, Doctoral Program, and Work*

in Progress at the Conference on Intelligent Computer Mathematics 2016 (CICM-WiP 2016), volume 1785 of *CEUR*, pages 219–228. CEUR-WS.org, 2016.

[Göd92] Kurt Gödel. *On formally undecidable propositions of Principia Mathematica and related systems*. Courier Corporation, 1992.

[Gon08] Georges Gonthier. Formal proof–the four-color theorem. *Notices of the AMS*, 55(11):1382–1393, 2008.

[GPAM+20] Ian Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Yoshua Bengio. Generative adversarial networks. *Communications of the ACM*, 63(11):139–144, 2020.

[GR14] Shixiang Gu and Luca Rigazio. Towards deep neural network architectures robust to adversarial examples, 2014.

[HAB+17] 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 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. *Forum of Mathematics, Pi*, 5, 2017.

[Häh01] Reiner Hähnle. Tableaux and related methods. In John Alan Robinson and Andrei Voronkov, editors, *Handbook of Automated Reasoning (in 2 volumes)*, pages 100–178. Elsevier and MIT Press, 2001.

[Hal08] Thomas C Hales. Formal proof. *Notices of the AMS*, 55(11):1370–1380, 2008.

[HKKS01] Christoph Helma, R King, S Kramer, and A. Srinivasan. The predictive toxicology challenge 2000-2001. *Bioinformatics*, 17, 01 2001.

[How69] William A. Howard. The formulae-as-types notion of construction. 1969.

[HR00] Michael Huth and Mark Dermot Ryan. *Logic in computer science - modelling and reasoning about systems*. Cambridge University Press, 2000.

[HS97] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural Computation*, 9:1735–1780, 1997.

[Hue02] Gérard P. Huet. Higher order unification 30 years later. In Victor Carreño, César A. Muñoz, and Sofiène Tahar, editors, *Theorem Proving in Higher Order Logics, 15th International Conference, TPHOLs 2002, Hampton, VA, USA, August 20-23, 2002, Proceedings*, volume 2410 of *Lecture Notes in Computer Science*, pages 3–12. Springer, 2002.

[HUW14] John Harrison, Josef Urban, and Freek Wiedijk. History of interactive theorem proving. In Jörg H. Siekmann, editor, *Computational Logic*, volume 9 of *Handbook of the History of Logic*, pages 135–214. Elsevier, 2014.

[HYL17] William L. Hamilton, Rex Ying, and Jure Leskovec. Inductive representation learning on large graphs, 2017.

[HZRS16] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016.

[JCO+20] Jan Jakubuv, Karel Chvalovský, Miroslav Olsák, Bartosz Piotrowski, Martin Suda, and Josef Urban. ENIGMA anonymous: Symbol-independent inference guiding machine (system description). 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 448–463. Springer, 2020.

[KAE+10] 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.

[KB14] Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

[KEI18] Tobias Kaminski, Thomas Eiter, and Katsumi Inoue. Exploiting answer set programming with external sources for meta-interpretive learning. *Theory and Practice of Logic Programming*, 18(3-4):571–588, October 2018.

[KHG12] Ekaterina Komendantskaya, Jónathan Heras, and Gudmund Grov. Machine learning in Proof General: Interfacing interfaces. In Cezary Kaliszyk and Christoph Lüth, editors, *Proceedings 10th International Workshop On User Interfaces for Theorem Provers, UITP 2012, Bremen, Germany, July 11th, 2012*, volume 118 of *EPTCS*, pages 15–41, 2012.

[KKM+16] Kristian Kersting, Nils M. Kriege, Christopher Morris, Petra Mutzel, and Marion Neumann. Benchmark data sets for graph kernels, 2016.

[Kra91] Mark A. Kramer. Nonlinear principal component analysis using autoassociative neural networks. *AIChE Journal*, 37(2):233–243, 1991.

[KS06] Levente Kocsis and Csaba Szepesvári. Bandit based monte-carlo planning. In *European conference on machine learning*, pages 282–293. Springer, 2006.

[KS11] Lukasz Kaiser and Lukasz Stafiniak. First-order logic with counting for general game playing. In Wolfram Burgard and Dan Roth, editors, *Proceedings of the Twenty-Fifth AAAI Conference on Artificial Intelligence, AAAI 2011, San Francisco, California, USA, August 7-11, 2011*. AAAI Press, 2011.

[KSH17] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. Imagenet classification with deep convolutional neural networks. *Commun. ACM*, 60(6):84–90, May 2017.

[KSUV15] Cezary Kaliszyk, Stephan Schulz, Josef Urban, and Jiří Vyskočil. System description: E.T. 0.1. In Amy P. Felty and Aart Middeldorp, editors, *Proc. of 25th International Conference on Automated Deduction (CADE'15)*, volume 9195 of *LNCS*, pages 389–398. Springer-Verlag, 2015.

[KU15] Cezary Kaliszyk and Josef Urban. MizAR 40 for Mizar 40. *J. Autom. Reasoning*, 55(3):245–256, 2015.

[KUMO18] Cezary Kaliszyk, Josef Urban, Henryk Michalewski, and Miroslav Olšák. Reinforcement learning of theorem proving. In S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett, editors, *Advances in Neural Information Processing Systems 31*, pages 8836–8847. Curran Associates, Inc., 2018.

[KUV15] Cezary Kaliszyk, Josef Urban, and Jiří Vyskočil. Efficient semantic features for automated reasoning over large theories. In Qiang Yang and Michael Wooldridge, editors, *Proc. of the 24th International Joint Conference on Artificial Intelligence (IJCAI'15)*, pages 3084–3090. AAAI Press, 2015.

[KvLT+12] Daniel Kühlwein, Twan van Laarhoven, Evgeni Tsivtsivadze, Josef Urban, and Tom Heskes. Overview and evaluation of premise selection techniques for large theory mathematics. In Bernhard Gramlich, Dale Miller, and Uli Sattler, editors, *Automated Reasoning*, pages 378–392, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.

[KVV13] Michael K. Kinyon, Robert Veroff, and Petr Vojtechovský. Loops with abelian inner mapping groups: An application of automated deduction. In Maria Paola Bonacina and Mark E. Stickel, editors, *Automated Reasoning and Mathematics - Essays in Memory of William W. McCune*, volume 7788 of *Lecture Notes in Computer Science*, pages 151–164. Springer, 2013.

[KW16] Thomas N Kipf and Max Welling. Semi-supervised classification with graph convolutional networks. *arXiv preprint arXiv:1609.02907*, 2016.

[LBH15] Yann LeCun, Y. Bengio, and Geoffrey Hinton. Deep learning. *Nature*, 521:436–44, 05 2015.

[Ler09] Xavier Leroy. Formal verification of a realistic compiler. *Commun. ACM*, 52(7):107–115, 2009.

[LHYG19] Haonan Lu, Seth H. Huang, Tian Ye, and Xiuyan Guo. Graph star net for generalized multi-task learning. *CoRR*, abs/1906.12330, 2019.

[Lif19] Vladimir Lifschitz. *Answer Set Programming*. Springer, August 2019.

[LISK17] Sarah Loos, Geoffrey Irving, Christian Szegedy, and Cezary Kaliszyk. Deep network guided proof search. In Thomas Eiter and David Sands, editors, *LPAR-21. 21st International Conference on Logic for Programming, Artificial Intelligence and Reasoning*, volume 46 of *EPiC Series in Computing*, pages 85–105. EasyChair, 2017.

[Llo87] John W. Lloyd. *Foundations of Logic Programming, 2nd Edition*. Springer, 1987.

[LRB14] Mark Law, Alessandra Russo, and Krysia Broda. Inductive learning of answer set programs. In *Proceedings of Logics in Artificial Intelligence*, pages 311–325. Springer, August 2014.

[MLPT14] Stephen H. Muggleton, Dianhuan Lin, Niels Pahlavi, and Alireza Tamaddoni-Nezhad. Meta-interpretive learning: application to grammatical inference. *Machince Learning*, 94(1):25–49, January 2014.

[MRP+12] Stephen H. Muggleton, Luc De Raedt, David Poole, Ivan Bratko, Peter A. Flach, Katsumi Inoue, and Ashwin Srinivasan. ILP turns 20 - biography and future challenges. *Mach. Learn.*, 86(1):3–23, 2012.

[MSC+13] Tomas Mikolov, Ilya Sutskever, Kai Chen, Greg Corrado, and Jeffrey Dean. Distributed representations of words and phrases and their compositionality, 2013.

[MSRR19] Ryan L Murphy, Balasubramaniam Srinivasan, Vinayak Rao, and Bruno Ribeiro. Relational pooling for graph representations. *arXiv preprint arXiv:1903.02541*, 2019.

[Mug91] Stephen Muggleton. Inductive logic programming. *New Generation Computing*, 8(4):295–318, February 1991.

[Mug95] Stephen Muggleton. Inverse entailment and progol. *New Generation Computing*, 13(3&4):245–286, December 1995.

[Nag19] Yutaka Nagashima. LiFtEr: Language to encode induction heuristics for Isabelle/HOL. In Anthony Widjaja Lin, editor, *Programming Languages and Systems - 17th Asian Symposium, APLAS 2019, Nusa Dua, Bali, Indonesia, December 1-4, 2019, Proceedings*, volume 11893 of *Lecture Notes in Computer Science*, pages 266–287. Springer, 2019.

[NAGA+21] Maxwell Nye, Anders Andreassen, Guy Gur-Ari, Henryk Michalewski, Jacob Austin, David Bieber, David Dohan, Aitor Lewkowycz, Maarten Bosma, David Luan, Charles Sutton, and Augustus Odena. Show your work: Scratchpads for intermediate computation with language models. *ArXiv*, abs/2112.00114, 2021.

[NCWSC97] Shan-Hwei Nienhuys-Cheng, Ronald de Wolf, J. Siekmann, and J. G. Carbonell. *Foundations of Inductive Logic Programming*. Springer, 1997.

[NPW02] Tobias Nipkow, Lawrence C Paulson, and Markus Wenzel. *Isabelle/HOL: a proof assistant for higher-order logic*, volume 2283. Springer Science & Business Media, 2002.

[OdPPR19] Carlos Olarte, Valeria de Paiva, Elaine Pimentel, and Giselle Reis. The illtp library for intuitionistic linear logic. *arXiv preprint arXiv:1904.06850*, 2019.

[OdPPR20] Carlos Olarte, Valeria de Paiva, Elaine Pimentel, and Giselle Reis. Linear logic theorem proving. `https://github.com/meta-logic/lltp`, 2020.

[OKU19] Miroslav Olšák, Cezary Kaliszyk, and Josef Urban. Property invariant embedding for automated reasoning, 2019.

[OKU20] Miroslav Olsák, Cezary Kaliszyk, and Josef Urban. Property invariant embedding for automated reasoning. In Giuseppe De Giacomo, Alejandro Catalá, Bistra Dilkina, Michela Milano, Senén Barro, Alberto Bugarín, and Jérôme Lang, editors, *ECAI 2020 - 24th European Conference on Artificial Intelligence*, volume 325 of *Frontiers in Artificial Intelligence and Applications*, pages 1395–1402. IOS Press, 2020.

[PAK20] Julian Parsert, Stephanie Autherith, and Cezary Kaliszyk. Property preserving embedding of first-order logic. In *GCAI*, volume 72 of *EPiC Series in Computing*, pages 70–82. EasyChair, 2020.

[PCK22a] Stanisław J. Purgał, David M. Cerna, and C. Kaliszyk. Differentiable inductive logic programming in high-dimensional space. *ArXiv*, abs/2208.06652, 2022.

[PCK22b] Stanisław J. Purgał, David M. Cerna, and Cezary Kaliszyk. Learning higher-order logic programs from failures. In Lud De Raedt, editor, *Proceedings of the Thirty-First International Joint Conference on Artificial Intelligence, IJCAI-22*, pages 2726–2733. International Joint Conferences on Artificial Intelligence Organization, 7 2022. Main Track.

[PGM+19] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito,

Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. Pytorch: An imperative style, high-performance deep learning library. In *Advances in Neural Information Processing Systems 32*, pages 8024–8035. Curran Associates, Inc., 2019.

[PK22] Stanisław J. Purgał and Cezary Kaliszyk. Adversarial learning to reason in an arbitrary logic. *The International FLAIRS Conference Proceedings*, 35, May 2022.

[Plo70] Gordon D. Plotkin. A note on inductive generalization. *Machine Intelligence*, 5(1):153–163, 1970.

[PLR+20] Aditya Sanjay Paliwal, Sarah M. Loos, Markus N. Rabe, Kshitij Bansal, and Christian Szegedy. Graph representations for higher-order logic and theorem proving. *ArXiv*, abs/1905.10006, 2020.

[PPK21] Stanisław Purgał, Julian Parsert, and Cezary Kaliszyk. A study of continuous vector representations for theorem proving. *Journal of Logic and Computation*, 02 2021.

[Pur20] Stanislaw J. Purgal. Improving expressivity of graph neural networks. *2020 International Joint Conference on Neural Networks (IJCNN)*, pages 1–7, 2020.

[QCJ93] J. R. Quinlan and R. M. Cameron-Jones. Foil: A midterm report. In *Proceedings of the European Conference on Machine Learning*, pages 1–20, Vienna, Austria, April 1993. Springer.

[Qui90] J. Ross Quinlan. Learning logical definitions from relations. *Machine Learning*, 5:239–266, August 1990.

[Rae08] Luc De Raedt. *Logical and relational learning*. Cognitive Technologies. Springer, 2008.

[Rey70] John C. Reynolds. Transformational systems and the algebraic structure of atomic formulas. *Machine Intelligence*, 5(1):135–151, 1970.

[RKB+22] Laura Ruis, Akbir Khan, Stella Rose Biderman, Sara Hooker, Tim Rocktaschel, and Edward Grefenstette. Large language models are not zero-shot communicators. *ArXiv*, abs/2210.14986, 2022.

[RMBL19] Emanuele Rossi, Federico Monti, Michael Bronstein, and Pietro Liò. ncrna classification with graph convolutional networks. *arXiv preprint arXiv:1905.06515*, 2019.

[RO11] Thomas Raths and Jens Otten. The qmltp library: Benchmarking theorem provers for modal logics. Technical report, Technical Report, University of Potsdam, 2011.

[ROK05] Thomas Raths, Jens Otten, and Christoph Kreitz. The iltp library: Benchmarking automated theorem provers for intuitionistic logic. In *International Conference on Automated Reasoning with Analytic Tableaux and Related Methods*, pages 333–337. Springer, 2005.

[RR19] Michael Rawson and Giles Reger. A neurally-guided, parallel theorem prover. In Andreas Herzig and Andrei Popescu, editors, *Frontiers of Combining Systems - 12th International Symposium, FroCoS 2019, London, UK, September 4-6, 2019, Proceedings*, volume 11715 of *Lecture Notes in Computer Science*, pages 40–56. Springer, 2019.

[RV01] John Alan Robinson and Andrei Voronkov, editors. *Handbook of Automated Reasoning (in 2 volumes)*. Elsevier and MIT Press, 2001.

[RV02] Alexandre Riazanov and Andrei Voronkov. The design and implementation of vampire. *AI Commun.*, 15(2,3):91–110, aug 2002.

[RW10] Bertrand Russell and Alfred North Whitehead. Principia mathematica vol. i. 1910.

[RWC+19] Alec Radford, Jeff Wu, Rewon Child, David Luan, Dario Amodei, and Ilya Sutskever. Language models are unsupervised multitask learners. 2019.

[SCV19] Stephan Schulz, Simon Cruanes, and Petar Vukmirović. Faster, higher, stronger: E 2.3. In Pascal Fontaine, editor, *Proc. of the 27th CADE, Natal, Brasil*, number 11716 in LNAI, pages 495–507. Springer, 2019.

[SFA+22] Teven Le Scao, Angela Fan, Christopher Akiki, Elizabeth-Jane Pavlick, Suzana Ili'c, Daniel Hesslow, Roman Castagn'e, Alexandra Sasha Luccioni, Franccois Yvon, Matthias Gallé, Jonathan Tow, Alexander M. Rush, Stella Rose Biderman, Albert Webson, Pawan Sasanka Ammanamanchi, Thomas Wang, Benoît Sagot, Niklas Muennighoff, Albert Villanova del Moral, Olatunji Ruwase, Rachel Bawden, Stas Bekman, Angelina McMillan-Major, Iz Beltagy, Huu Nguyen, Lucile Saulnier, Samson Tan, Pedro Ortiz Suarez, Victor Sanh, Hugo Laurenccon, Yacine Jernite, Julien Launay, Margaret Mitchell, Colin Raffel, Aaron Gokaslan, Adi Simhi, Aitor Soroa Etxabe, Alham Fikri Aji, Amit Alfassy, Anna Rogers, Ariel Kreisberg Nitzav, Canwen Xu, Chenghao Mou, Chris C. Emezue, Christopher Klamm, Colin Leong, Daniel Alexander van Strien, David Ifeoluwa Adelani, Dragomir R. Radev, Eduardo G. Ponferrada, Efrat Levkovizh, Ethan Kim, Eyal Bar Natan, Francesco De Toni, Gérard Dupont, Germán Kruszewski, Giada Pistilli, Hady ElSahar, Hamza Benyamina, Hieu Tran, Ian Yu, Idris Abdulmumin, Isaac Johnson, Itziar Gonzalez-Dios, Javier de la Rosa, Jenny Chim, Jesse Dodge, Jian Zhu, Jonathan Chang, Jorg Frohberg, Josephine L. Tobing, Joydeep Bhattacharjee, Khalid Almubarak, Kimbo Chen, Kyle Lo, Leandro von Werra, Leon Weber, Long Phan, Loubna Ben Allal, Ludovic

Tanguy, Manan Dey, Manuel Romero Muñoz, Maraim Masoud, Mar'ia Grandury, Mario vSavsko, Max Huang, Maximin Coavoux, Mayank Singh, Mike Tian-Jian Jiang, Minh Chien Vu, Mohammad Ali Jauhar, Mustafa Ghaleb, Nishant Subramani, Nora Kassner, Nurulaqilla Khamis, Olivier Nguyen, Omar Espejel, Ona de Gibert, Paulo Villegas, Peter Henderson, Pierre Colombo, Priscilla Amuok, Quentin Lhoest, Rheza Harliman, Rishi Bommasani, Roberto L'opez, R. Ribeiro, Salomey Osei, Sampo Pyysalo, Sebastian Nagel, Shamik Bose, Shamsuddeen Hassan Muhammad, Shanya Sharma, S. Longpre, Somaieh Nikpoor, Stanislav Silberberg, Suhas Pai, Sydney Zink, Tiago Timponi Torrent, Timo Schick, Tristan Thrush, Valentin Danchev, Vassilina Nikoulina, Veronika Laippala, Violette Lepercq, Vrinda Prabhu, Zaid Alyafeai, Zeerak Talat, Arun Raja, Benjamin Heinzerling, Chenglei Si, Elizabeth Salesky, Sabrina J. Mielke, Wilson Y. Lee, Abheesht Sharma, Andrea Santilli, Antoine Chaffin, Arnaud Stiegler, Debajyoti Datta, Eliza Szczechla, Gunjan Chhablani, Han Wang, Harshit Pandey, Hendrik Strobelt, Jason Alan Fries, Jos Rozen, Leo Gao, Lintang Sutawika, M Saiful Bari, Maged S. Al-shaibani, Matteo Manica, Nihal V. Nayak, Ryan Teehan, Samuel Albanie, Sheng Shen, Srulik Ben-David, Stephen H. Bach, Taewoon Kim, Tali Bers, Thibault Févry, Trishala Neeraj, Urmish Thakker, Vikas Raunak, Xiang Tang, Zheng Xin Yong, Zhiqing Sun, Shaked Brody, Y Uri, Hadar Tojarieh, Adam Roberts, Hyung Won Chung, Jaesung Tae, Jason Phang, Ofir Press, Conglong Li, Deepak Narayanan, Hatim Bourfoune, Jared Casper, Jeff Rasley, Max Ryabinin, Mayank Mishra, Minjia Zhang, Mohammad Shoeybi, Myriam Peyrounette, Nicolas Patry, Nouamane Tazi, Omar Sanseviero, Patrick von Platen, Pierre Cornette, Pierre Franccois Lavall'ee, Rémi Lacroix, Samyam Rajbhandari, Sanchit Gandhi, Shaden Smith, Stéphane Requena, Suraj Patil, Tim Dettmers, Ahmed Baruwa, Amanpreet Singh, Anastasia Cheveleva, Anne-Laure Ligozat, Arjun Subramonian, Aur'elie N'ev'eol, Charles Lovering, Daniel H Garrette, Deepak R. Tunuguntla, Ehud Reiter, Ekaterina Taktasheva, Ekaterina Voloshina, Eli Bogdanov, Genta Indra Winata, Hailey Schoelkopf, Jan-Christoph Kalo, Jekaterina Novikova, Jessica Zosa Forde, Jordan Clive, Jungo Kasai, Ken Kawamura, Liam Hazan, Marine Carpuat, Miruna Clinciu, Najoung Kim, Newton Cheng, Oleg Serikov, Omer Antverg, Oskar van der Wal, Rui Zhang, Ruochen Zhang, Sebastian Gehrmann, S. Osher Pais, Tatiana Shavrina, Thomas Scialom, Tian Yun, Tomasz Limisiewicz, Verena Rieser, Vitaly Protasov, Vladislav Mikhailov, Yada Pruksachatkun, Yonatan Belinkov, Zachary Bamberger, Zdenvek Kasner, Alice Rueda, Amanda Pestana, Amir Feizpour, Ammar Khan, Amy Faranak, Ananda Santa Rosa Santos, Anthony Hevia, Antigona Unldreaj, Arash Aghagol, Arezoo Abdollahi, Aycha Tammour, Azadeh HajiHosseini, Bahareh Behroozi, Benjamin Olusola Ajibade, Bharat Kumar Saxena, Carlos Muñoz Ferrandis, Danish Contractor, David M. Lansky, Davis David, Douwe Kiela, Duong Anh Nguyen, Edward Tan, Emily Baylor, Ezinwanne Ozoani, Fatim T Mirza, Frankline

Ononiwu, Habib Rezanejad, H.A. Jones, Indrani Bhattacharya, Irene Solaiman, Irina Sedenko, Isar Nejadgholi, J. Lawrence Passmore, Joshua Seltzer, Julio Bonis Sanz, Karën Fort, Lívia Macedo Dutra, Mairon Samagaio, Maraim Elbadri, Margot Mieskes, Marissa Gerchick, Martha Akinlolu, Michael McKenna, Mike Qiu, M. K. K. Ghauri, Mykola Burynok, Nafis Abrar, Nazneen Rajani, Nour Elkott, Nourhan Fahmy, Olanrewaju Modupe Samuel, Ran An, R. P. Kromann, Ryan Hao, Samira Alizadeh, Sarmad Shubber, Silas L. Wang, Sourav Roy, Sylvain Viguier, Thanh-Cong Le, Tobi Oyebade, Trieu Nguyen Hai Le, Yoyo Yang, Zachary Kyle Nguyen, Abhinav Ramesh Kashyap, Alfredo Palasciano, Alison Callahan, Anima Shukla, Antonio Miranda-Escalada, Ayush Kumar Singh, Benjamin Beilharz, Bo Wang, Caio Matheus Fonseca de Brito, Chenxi Zhou, Chirag Jain, Chuxin Xu, Clémentine Fourrier, Daniel Le'on Perin'an, Daniel Molano, Dian Yu, Enrique Manjavacas, Fabio Barth, Florian Fuhrimann, Gabriel Altay, Giyaseddin Bayrak, Gully A. Burns, Helena U. Vrabec, Iman I.B. Bello, Isha Dash, Ji Soo Kang, John Giorgi, Jonas Golde, Jose David Posada, Karthi Sivaraman, Lokesh Bulchandani, Lu Liu, Luisa Shinzato, Madeleine Hahn de Bykhovetz, Maiko Takeuchi, Marc Pàmies, María Andrea Castillo, Marianna Nezhurina, Mario Sanger, Matthias Samwald, Michael Cullan, Michael Weinberg, M Wolf, Mina Mihaljcic, Minna Liu, Moritz Freidank, Myungsun Kang, Natasha Seelam, Nathan Dahlberg, Nicholas Michio Broad, Nikolaus Muellner, Pascale Fung, Patricia Haller, R. Chandrasekhar, R. Eisenberg, Robert Martin, Rodrigo L. Canalli, Rosaline Su, Ruisi Su, Samuel Cahyawijaya, Samuele Garda, Shlok S Deshmukh, Shubhanshu Mishra, Sid Kiblawi, Simon Ott, Sinee Sang-aroonsiri, Srishti Kumar, Stefan Schweter, Sushil Pratap Bharati, T. A. Laud, Th'eo Gigant, Tomoya Kainuma, Wojciech Kusa, Yanis Labrak, Yashasvi Bajaj, Y. Venkatraman, Yifan Xu, Ying Xu, Yun chao Xu, Zhee Xao Tan, Zhongli Xie, Zifan Ye, Mathilde Bras, Younes Belkada, and Thomas Wolf. Bloom: A 176b-parameter open-access multilingual language model. *ArXiv*, abs/2211.05100, 2022.

[SGT+08] Franco Scarselli, Marco Gori, Ah Chung Tsoi, Markus Hagenbuchner, and Gabriele Monfardini. The graph neural network model. *IEEE Transactions on Neural Networks*, 20(1):61–80, 2008.

[SHK+14] Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout: a simple way to prevent neural networks from overfitting. *The journal of machine learning research*, 15(1):1929–1958, 2014.

[SHM+16] 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, 2016.

[SHS+17] David Silver, Thomas Hubert, Julian Schrittwieser, Ioannis Antonoglou, Matthew Lai, Arthur Guez, Marc Lanctot, Laurent Sifre, Dharshan Kumaran, Thore Graepel, Timothy P. Lillicrap, Karen Simonyan, and Demis Hassabis. Mastering chess and shogi by self-play with a general reinforcement learning algorithm. *ArXiv*, abs/1712.01815, 2017.

[SHS+18] David Silver, Thomas Hubert, Julian Schrittwieser, Ioannis Antonoglou, Matthew Lai, Arthur Guez, Marc Lanctot, Laurent Sifre, Dharshan Kumaran, Thore Graepel, et al. A general reinforcement learning algorithm that masters chess, shogi, and go through self-play. *Science*, 362(6419):1140–1144, 2018.

[SKB03] Ashwin Srinivasan, Ross D. King, and Michael E. Bain. An empirical study of the use of relevance information in inductive logic programming. *Journal of Machine Learning Research*, 4:369–383, December 2003.

[SL08] Armando Solar-Lezama. *Program Synthesis by Sketching*. PhD thesis, University of California at Berkeley, USA, December 2008.

[SMDH13] Ilya Sutskever, James Martens, George Dahl, and Geoffrey Hinton. On the importance of initialization and momentum in deep learning. In *Proceedings of the 30th International Conference on International Conference on Machine Learning - Volume 28*, ICML'13, page III–1139–III–1147. JMLR.org, 2013.

[SP97] M. Schuster and K. K. Paliwal. Bidirectional recurrent neural networks. *IEEE Transactions on Signal Processing*, 45(11):2673–2681, 1997.

[Sri01] Ashwin Srinivasan. The ALEPH manual. Technical report, Machine Learning at the Computing Laboratory, Oxford University, 2001.

[SSS+17] David Silver, Julian Schrittwieser, Karen Simonyan, Ioannis Antonoglou, Aja Huang, Arthur Guez, Thomas Hubert, Lucas Baker, Matthew Lai, Adrian Bolton, et al. Mastering the game of go without human knowledge. *nature*, 550(7676):354–359, 2017.

[Sut17] G. Sutcliffe. The TPTP Problem Library and Associated Infrastructure. From CNF to TH0, TPTP v6.4.0. *Journal of Automated Reasoning*, 59(4):483–502, 2017.

[TSM15] Kai Sheng Tai, Richard Socher, and Christopher D. Manning. Improved semantic representations from tree-structured long short-term memory networks. *Proceedings of the 53rd Annual Meeting of the Association for Computational Linguistics and the 7th International Joint Conference on Natural Language Processing (Volume 1: Long Papers)*, 2015.

[UVŠ11] Josef Urban, Jiří Vyskočil, and Petr Štěpánek. MaLeCoP machine learning connection prover. In Kai Brünnler and George Metcalfe, editors, *Automated Reasoning with Analytic Tableaux and Related Methods*, pages 263–277, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg.

[VCC+17] Petar Velickovic, Guillem Cucurull, Arantxa Casanova, Adriana Romero, Pietro Liò, and Yoshua Bengio. Graph attention networks. *ArXiv*, abs/1710.10903, 2017.

[vdODZ+16] Aaron van den Oord, Sander Dieleman, Heiga Zen, Karen Simonyan, Oriol Vinyals, Alex Graves, Nal Kalchbrenner, Andrew Senior, and Koray Kavukcuoglu. Wavenet: A generative model for raw audio, 2016.

[Vor14] Andrei Voronkov. AVATAR: the architecture for first-order theorem provers. In Armin Biere and Roderick Bloem, editors, *Computer Aided Verification - 26th International Conference, CAV 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 18-22, 2014. Proceedings*, volume 8559 of *Lecture Notes in Computer Science*, pages 696–710. Springer, 2014.

[VSP+17] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need. *CoRR*, abs/1706.03762, 2017.

[Wad91] William W. Wadge. Higher-order horn logic programming. In Vijay A. Saraswat and Kazunori Ueda, editors, *Proceedings of the International Symposium on Logic Programming*, pages 289–303, San Diego, California, USA, October 1991. MIT Press.

[WBKU20] 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.

[Wil92] Ronald J. Williams. Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Mach. Learn.*, 8:229–256, 1992.

[WKU18] 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, *11th International Conference on Intelligent Computer Mathematics (CICM 2018)*, volume 11006 of *LNCS*, pages 255–270. Springer, 2018.

[WL68] Boris Weisfeiler and Andrei A Lehman. A reduction of a graph to a canonical form and an algebra arising during this reduction. *Nauchno-Technicheskaya Informatsia*, 2(9):12–16, 1968.

[WSTL10] Jan Wielemaker, Tom Schrijvers, Markus Triska, and Torbjörn Lager. Swi-prolog, 2010.

[WTWD17] 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.

[Wu19] David J. Wu. Accelerating self-play learning in go. *CoRR*, abs/1902.10565, 2019.

[XHLJ18] Keyulu Xu, Weihua Hu, Jure Leskovec, and Stefanie Jegelka. How powerful are graph neural networks? *arXiv preprint arXiv:1810.00826*, 2018.

[XWZ+19] Qi Xuan, Jinhuan Wang, Minghao Zhao, Junkun Yuan, Chenbo Fu, Zhongyuan Ruan, and Guanrong Chen. Subgraph networks with application to structural feature space expansion. *IEEE Transactions on Knowledge and Data Engineering*, 2019.

[Yan] Xifeng Yan. Graph datasets.

[YBY+19] Zhitao Ying, Dylan Bourgeois, Jiaxuan You, Marinka Zitnik, and Jure Leskovec. Gnnexplainer: Generating explanations for graph neural networks. In *Advances in Neural Information Processing Systems*, pages 9240–9251, 2019.

[YJK+19] Seongjun Yun, Minbyul Jeong, Raehyun Kim, Jaewoo Kang, and Hyunwoo J Kim. Graph transformer networks. In *Advances in Neural Information Processing Systems*, pages 11960–11970, 2019.

[YM18] Abdou Youssef and Bruce R. Miller. Deep learning for math knowledge processing. 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 271–286. Springer, 2018.

[ZNL+22] Hattie Zhou, Azade Nova, H. Larochelle, Aaron C. Courville, Behnam Neyshabur, and Hanie Sedghi. Teaching algorithmic reasoning via in-context learning. *ArXiv*, abs/2211.09066, 2022.

[ZRG+22] Susan Zhang, Stephen Roller, Naman Goyal, Mikel Artetxe, Moya Chen, Shuohui Chen, Christopher Dewan, Mona Diab, Xian Li, Xi Victoria Lin, Todor Mihaylov, Myle Ott, Sam Shleifer, Kurt Shuster, Daniel Simig, Punit Singh Koura, Anjali Sridhar, Tianlu Wang, and Luke Zettlemoyer. Opt: Open pre-trained transformer language models. *ArXiv*, abs/2205.01068, 2022.

[ZXC+18] Wenting Zhao, Chunyan Xu, Zhen Cui, Tong Zhang, Jiatao Jiang, Zhenyu Zhang, and Jian Yang. When work matters: Transforming classical network structures to graph cnn. *arXiv preprint arXiv:1807.02653*, 2018.