A Study of Continuous Vector Representations for Theorem Proving

Stanisław Purgał[1], Julian Parsert[2], and Cezary Kaliszyk[1,3]

[1] University of Innsbruck, Innsbruck, Austria
{stanislaw.purgal,cezary.kaliszyk}@uibk.ac.at
[2] University of Oxford, Oxford, UK
julian.parsert@gmail.com
[3] University of Warsaw, Poland

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, 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.

1 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 a verification of proofs with the highest level of scrutiny, as well as an 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 autoembedding 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 2 we introduce the logical and machine learning preliminaries. In Section 3 we discuss related work. In Section 4 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 5 we develop a logical properties dataset and present the Mizar40 dataset. Section 6 contains an experimental evaluation of our approach. Finally Section 7 concludes and gives an outlook on the future work.

2 Preliminaries

2.1 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 [HR04].

An abstract Backus-Naur Form (BNF) for FOL formulas is presented below. The two main concepts are terms (1) and formulas (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.

$$\text{term} := \text{var} \mid \text{const} \mid f(\text{term}, \ldots, \text{term}) \tag{1}$$

$$\begin{aligned} \text{formula} := {} & \text{Atom}(\text{term}, \ldots, \text{term}) \\ & \mid \neg\text{formula} \mid \text{formula} \wedge \text{forumla} \\ & \mid \text{formula} \vee \text{formula} \\ & \mid \text{formula} \rightarrow \text{formula} \mid \text{formula} \leftrightarrow \text{formula} \\ & \mid \exists \text{ var. formula} \mid \forall \text{ var. formula} \end{aligned} \tag{2}$$

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 (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][4]. 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]:`

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

`![F]: (disjoint(D,F) <=> ~intersect(D,F))` which corresponds to the formula $\forall d.\, \forall f.\ \mathrm{disjoint}(d, f) \iff \neg\,\mathrm{intersect}(d, f)$. As part of the data extraction, we developed a parser for TPTP formulas where we took some liberties. 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*.

2.2 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 1) 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 1.

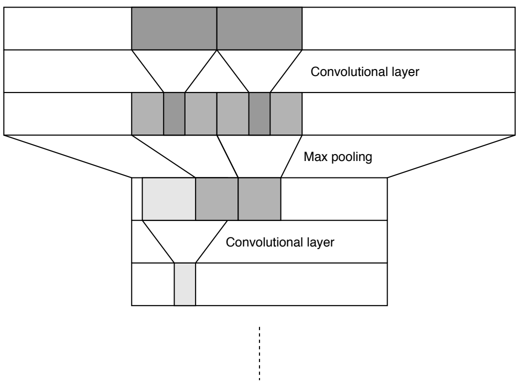

Fig. 1. Convolutional network

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.

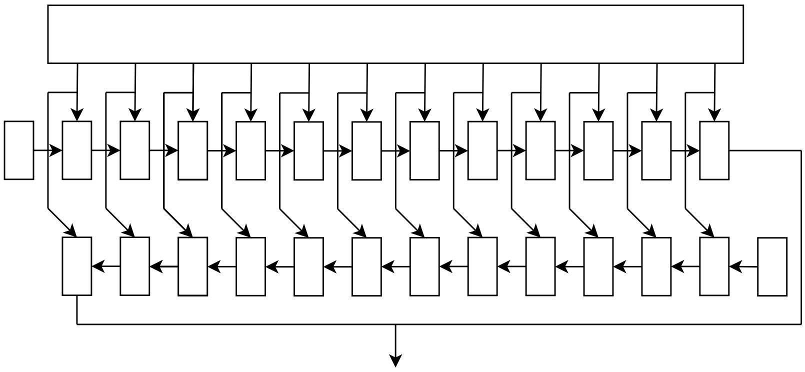

Fig. 2. Bidirectional LSTM network

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 neighbourhood, but from cells increasingly further away in the sequence. Figure 3 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].

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.

Autoencoders [Kra91] are neural networks trained to express identity function on some data. Their architecture usually contains some *bottleneck*, which forces the

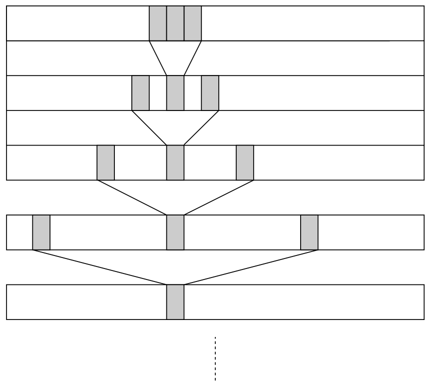

Fig. 3. WaveNet network

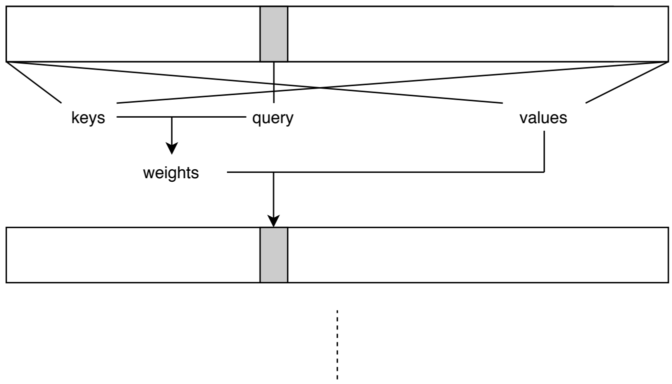

Fig. 4. Transformer encoder network

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.

3 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 neighbouring 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 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 6.2) 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.

4.1 Explicit Approach

The general setup for this approach is depicted in Figure 5. The green box in Figure 5 represents an encoding network for which we consider different models which we discuss later in this section. This network produces an encoding $\mathrm{enc}(\phi)$ of a formula ϕ. This continuous vector representation is then fed into classifiers that recognise logical properties (c.f. Section 5.1). 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 5.1. We train the network on this setup and evaluate the whole training setup (encoding network and classifiers) on unseen data in Section 6. However, it is important to note that we are only interested in the encoding network. Hence, we can extract the encoding

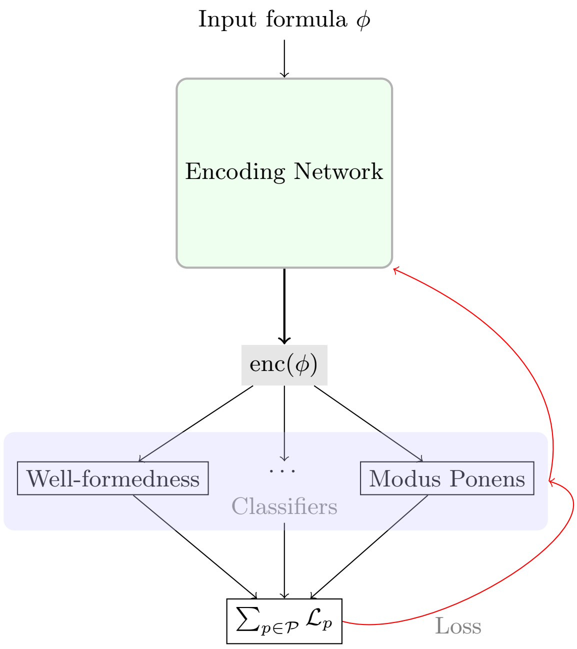

Fig. 5. The property training framework. The bottom area contains the classifiers that get one or more continuous representations of formulas enc(ϕ) as input. If the classifier takes two formulas as input (i.e. alpha-equivalence), we gather enc(ϕ_1) and enc(ϕ_2) separately and forward the pair (enc(ϕ_1), enc(ϕ_2)) to the classifier. The encoding networks are described subsequently (cf. Figure 6).

network (c.f. Figure 5) 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 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 6. The exact dimensions and sizes of the models are discussed in Section 6.

CNN based models The models based on CNNs are depicted on the left in Figure 6. 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 6 we evaluate these models.

LSTM based models The LSTM based models are depicted on the right side in Figure 6. 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.

4.2 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 7 depicts a high-level overview of this setup.

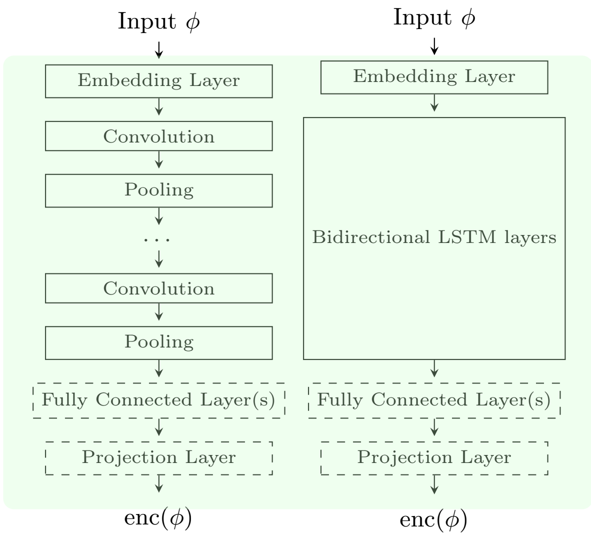

Fig. 6. The encoding models we considered with the layers that the input passes through. The left diagram depicts CNN-based models, while the right one depicts LSTM-based models. The dashed boxes describe layers that are optional for these model types.

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.

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 8).

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 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

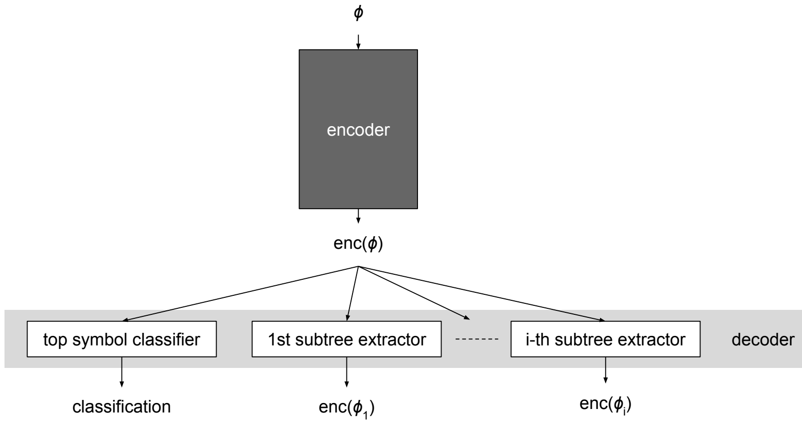

Fig. 7. Tree autoencoder mechanism

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.

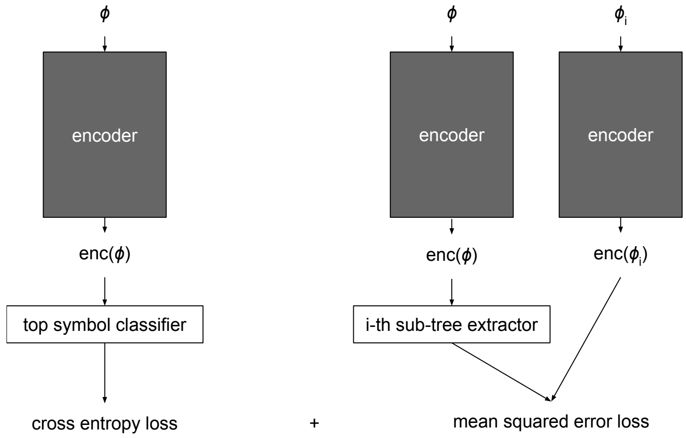

Fig. 8. Difference training mechanism

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 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 2).

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.

5 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

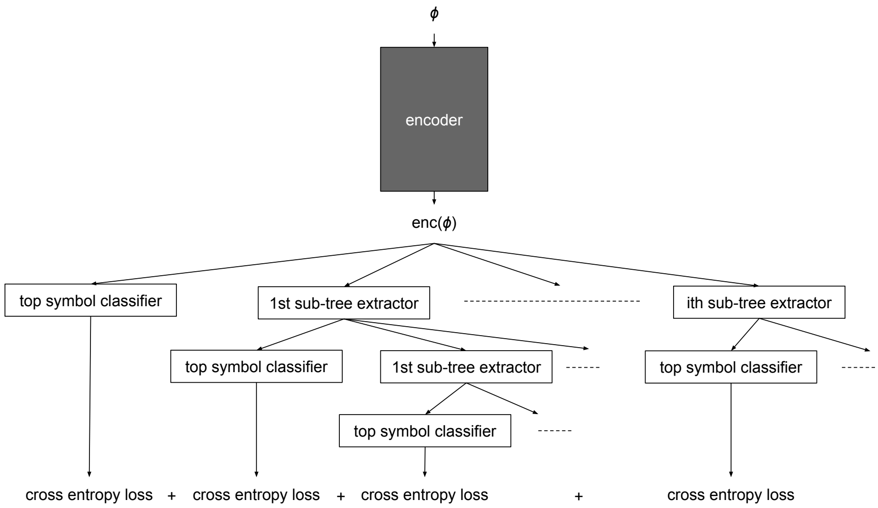

Fig. 9. Recursive training mechanism

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.

5.1 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:

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

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 recognising 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 (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.

$$\frac{\overline{P} \qquad \overline{P \rightarrow Q}}{Q} \tag{3}$$

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 1. We can associate the two formulas $\phi := \forall x.\ ((P(x) \rightarrow Q(x)) \wedge P(x))$ 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 synthesising 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 synthesise 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 1 and 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 2 showcases these concepts in more detail.

Example 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 realised 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.

5.2 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[5]. 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.

6 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 6.2 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.

6.1 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 2). Here, we have two different ways of obtaining evaluation and test data. We also want the encoding networks to generalise 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.

For the first and more straightforward evaluation, we use the data extracted dataset from the Graph Theory and Set Theory library described in Section 5.1 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

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

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 1. The multi-label

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 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)

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[6]. 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

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

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 additional fully connected layers has no negative effect. This leads us to distinguish two types of the 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 recognise 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 detected 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 4.1 we get better results for simple properties such as the presence of a conjunction.

6.2 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 neighbouring 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.

		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 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.

Decoding accuracy After training the autoencoders (Figure 11) 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

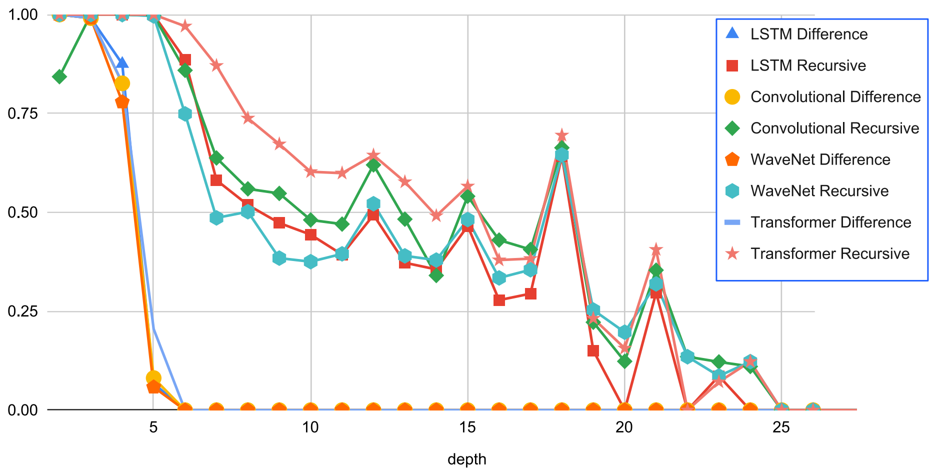

Fig. 10. Decoding accuracy of formulas over formula depth in the logical properties dataset.

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 2. 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 10 shows a plot of the decoding accuracy as the depth of the formula increases. Unsurprisingly, for very shallow formulas both types of networks 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 5.1. 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 3. 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 10).

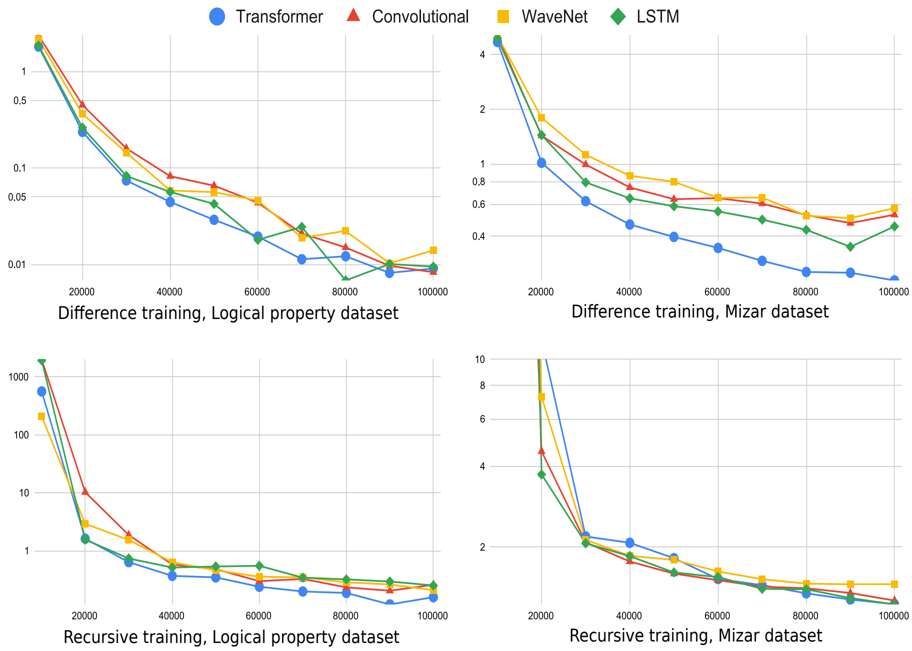

Fig. 11. Loss during training. The top two graphs present loss during difference training, and the bottom two graphs during recursive training. Note a different vertical scale for the four graphs, this is because the losses for the different training modes and datasets are hard to compare, however all four converge well.

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 5.2). The experiment (as described in Section 6.2) 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. Our general 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).

7 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

		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 3. Logical property classification accuracy on test set.

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 recognised, 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 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 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. Premise selection accuracy on test set.

difference training and the other recursive training. In order to train and to 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 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*.

References

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, Czesław Byliński, Adam Grabowski, Artur Korniłowicz, Roman Matuszewski, Adam Naumowicz, and Karol Pąk. The role of the Mizar Mathematical Library for interactive proof development in Mizar. *J. Autom. Reasoning*, 61(1-4):9–32, 2018.

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.

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

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

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.

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.

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.

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.

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

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.

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.

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

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.

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

HR04. Michael Huth and Mark Dermot Ryan. *Logic in computer science - modelling and reasoning about systems*. Cambridge University Press, 2004.

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.

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, 2014.

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.

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

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. *Journal of Automated Reasoning*, 55:245–256, 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, 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.

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

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.

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

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.

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

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.

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.

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.

SCV19. Stephan Schulz, Simon Cruanes, and Petar Vukmirovic. Faster, higher, stronger: E 2.3. 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 495–507. Springer, 2019.

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

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.

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, 2017.

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.

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.

WTWD17. Mingzhe Wang, Yihe Tang, Jian Wang, and Jia Deng. Premise selection for theorem proving by deep graph embedding. In Isabelle Guyon, Ulrike von Luxburg, Samy Bengio, Hanna M. Wallach, Rob Fergus, S. V. N. Vishwanathan, and Roman Garnett, editors, *Advances in Neural Information Processing Systems 30: Annual Conference on Neural Information Processing Systems 2017, 4-9 December 2017, Long Beach, CA, USA*, pages 2786–2796, 2017.

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.