Checking LTL Satisfiability via End-to-end Learning

Weilin Luo
School of Computer Science and Engineering, Sun Yat-sen University
Guangzhou, China
luowlin5@mail.sysu.edu.cn

Hai Wan*
School of Computer Science and Engineering, Sun Yat-sen University
Guangzhou, China
wanhai@mail.sysu.edu.cn

Delong Zhang
School of Computer Science and Engineering, Sun Yat-sen University
Guangzhou, China
zhangdlong3@mail2.sysu.edu.cn

Jianfeng Du*
Guangdong University of Foreign Studies
Pazhou Lab
Guangzhou, China
jfdu@gdufs.edu.cn

Hengdi Su
School of Computer Science and Engineering, Sun Yat-sen University
Guangzhou, China
suhd3@mail2.sysu.edu.cn

ABSTRACT

Linear temporal logic (LTL) satisfiability checking is a fundamental and hard (PSPACE-complete) problem. In this paper, we explore checking LTL satisfiability via end-to-end learning, so that we can take only polynomial time to check LTL satisfiability. Existing approaches have shown that it is possible to leverage end-to-end neural networks to predict the Boolean satisfiability problem with performance considerably higher than random guessing. Inspired by these approaches, we study two interesting questions: can end-to-end neural networks check LTL satisfiability, and can neural networks capture the semantics of LTL? To this end, we train different neural networks for keeping three logical properties of LTL, *i.e.*, recursive property, permutation invariance, and sequentiality. We demonstrate that neural networks can indeed capture some effective biases for checking LTL satisfiability. Besides, designing a special neural network keeping the logical properties of LTL can provide a better inductive bias. We also show the competitive results of neural networks compared with state-of-the-art approaches, *i.e.*, nuXmv and Aalta, on large scale datasets.

CCS CONCEPTS

• **Theory of computation** → **Modal and temporal logics**; • **Computing methodologies** → **Neural networks**.

KEYWORDS

linear temporal logic, satisfiability checking, neural networks

ACM Reference Format:
Weilin Luo, Hai Wan, Delong Zhang, Jianfeng Du, and Hengdi Su. 2022. Checking LTL Satisfiability via End-to-end Learning . In *37th IEEE/ACM International Conference on Automated Software Engineering (ASE '22), October 10–14, 2022, Rochester, MI, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3551349.3561163

*Both authors are corresponding authors.

ASE '22, October 10–14, 2022, Rochester, MI, USA

ACM ISBN 978-1-4503-9475-8/22/10…$15.00
https://doi.org/10.1145/3551349.3561163

1 INTRODUCTION

Linear temporal logic (LTL) [47] is one of the fundamental modal logics widely used in software engineering. Checking LTL satisfiability aims to answer whether a given LTL formula is satisfiable or unsatisfiable. It plays a core role in many problems, such as model checking [17], goal-conflict analysis [16, 39], and business process [42]. However, LTL satisfiability checking is PSPACE-complete [61]. Due to the high complexity challenge and wide applications, LTL satisfiability checking has attracted a fair amount of attention over the past few years. Developing new approaches over different paradigms remains to be an interesting and necessary research direction.

Existing approaches are based on logical reasoning mechanisms, such as model checking [50, 51], tableau [5, 29, 57, 66], temporal resolution [21, 53], and anti-chain [67]. Li et al. (2013) reduced LTL satisfiability checking to reachability analysis of a transition system [35]. This allows them to check LTL satisfiability on-the-fly, so that they can quickly find a satisfiable solution. Further, they proposed a series of SAT-based approaches [32, 34, 36–38] (Boolean satisfiability problem is denoted by SAT for short) to further improve the performance. We refer to these approaches as *logical approaches*, which are sound and complete, *i.e.*, guarantee correctly answering whether a given formula is satisfiable or not. However, all of them still suffer from the efficiency problem. Especially for LTL-SAT-heavy tasks such as goal-conflict identification [16, 39], the efficiency of checking LTL satisfiability is the main bottleneck.

Recently, some work [11, 59] has attempted to train end-to-end neural networks to solve SAT problem, so that it can take only polynomial time to check the satisfiability. These methods achieve much better accuracies than random guessing, thanks to the neural networks that capture the *permutation invariance* of the Boolean formulae, *i.e.*, the satisfiability of formulae does not change for reordered variables and clauses. Although they are not competitive with state-of-the-art (SOTA) logical approaches, they have shown the potential of neural networks in solving computationally hard problems. Cameron et al. (2020) showed great progress with neural networks in solving random 3-SAT problems, a theoretically

intractable class of SAT problem, demonstrating that neural networks can get a highly confident solution in a very short time [11]. This motivates us to explore end-to-end neural networks for LTL satisfiability checking.

Although end-to-end neural networks cannot guarantee soundness and completeness, they are so efficient that they have wide potential applications. For example, in goal-conflict identification [16], neural networks can pre-filter out obviously invalid boundary conditions (BCs), which are LTL formulae satisfying multiple satisfiability constraints. We can extract knowledge from a highly accurate and confident neural network to guide the practice in LTL satisfiability checking. Besides, it is possible to design an SAT-verifiable neural network which gives a satisfiable trace as a proof if the formula is satisfiable. Details will be introduced in Section 3. In summary, all these applications benefit from the exploration of end-to-end neural networks for LTL satisfiability checking.

We consider two interesting questions: *can end-to-end neural networks check LTL satisfiability, and can neural networks capture the semantics of LTL?* The primary intuition behind the motivation for applying neural networks to the SAT problem is that aligning the properties of a neural network and a task helps the neural network learn the task [69]. Different from SAT checking of Boolean formulae, checking LTL satisfiability requires to design neural networks that are able to model the recursively defined syntax and semantics of LTL. The syntax of LTL is based on recursively building formulae from sub-formulae. For semantics, the satisfaction relation between trace and formula is defined on the satisfaction relation between sub-trace and sub-formula (Proposition 5.1). We call the property as *recursive property* of LTL. In addition, neural networks need to capture not only the *permutation invariance* of LTL, but also the *sequentiality*. On one hand, some logical operators of LTL are permutation invariant, *i.e.*, the satisfiability of the formula does not depend on the order of the sub-formulae connected by the operators. On the other hand, some logical operators of LTL are sequence-dependent. It is challenging for neural networks to model the above three properties to fit the semantics of LTL.

To capture the semantics of LTL, we explore the potential of three classes of neural networks for checking LTL satisfiability: sequence model, graph neural network (GNN) [27], and recursive neural network (TreeNN) [62, 63]. These three classes have varying degrees for keeping the logical properties of LTL. Experimental results on synthetic datasets show that TreeNN has clear advantages. To bridge the gap between the symbolic reasoning of LTL on discrete domains and neural network inference on continuous domains, TreeNN recursively aggregates and combines sub-tree features on the syntax tree, which naturally can characterize the recursive property of LTL. At the same time, it models different semantics of logical operators of LTL by learning independent combination functions and leveraging certain aggregation functions to satisfy the permutation invariance or sequentiality.

From the experimental results, we discover that neural networks can capture some inductive biases that favor classification. Not all neural networks perform well on tests of generalization ability across formula distributions. Only the neural networks keeping the logical properties of LTL can provide a better inductive bias.

We also evaluate the neural networks trained from synthetic datasets in predicting the satisfiability on some large scale datasets. The results demonstrate that TreeNN generalizes well across formula sizes and distributions and is competitive with SOTA logical approaches, *i.e.*, nuXmv and Aalta. In theory, the running time of a logical approach tends to grow exponentially with the size of the formula, while TreeNN keeps predicting the satisfiability in polynomial time. Although TreeNN does not guarantee correctness, it guarantees extremely short running time and yields highly confident results, which are worthwhile for LTL-SAT-heavy tasks. Our code and datasets are publicly available at https://github.com/chenpolong/TLNet.

2 BACKGROUND

2.1 Linear Temporal Logic

Linear temporal logic (LTL) [47] has been widely used to describe infinite behaviors. The syntax of LTL for a finite set of atomic propositions $\mathbb{P}$ includes the standard logical operators ($\vee$ and $\neg$) and temporal modal operators (*next* ($\bigcirc$) and *until* ($\mathcal{U}$)), given below:

$$\phi := p \mid \neg\varphi \mid \varphi' \wedge \varphi'' \mid \bigcirc\varphi \mid \varphi' \mathcal{U} \varphi'',$$

where $p \in \mathbb{P} \cup \{\top\}$ and φ, φ', and φ'' are LTL formulae. For brevity, we only consider the above fundamental operators in this paper. Operator *or* ($\vee$), *release* ($\mathcal{R}$), *eventually* ($\Diamond$), *always* ($\Box$), and *weak-until* ($\mathcal{W}$) are commonly used, and can be defined as $\varphi' \vee \varphi'' := \neg(\neg\varphi' \wedge \neg\varphi'')$, $\varphi' \mathcal{R} \varphi'' := \neg(\neg\varphi' \mathcal{U} \neg\varphi'')$, $\Diamond\varphi := \top \mathcal{U} \varphi$, $\Box\varphi := \neg(\top \mathcal{U} \neg\varphi)$, and $\varphi' \mathcal{W} \varphi'' := \varphi' \mathcal{U} (\varphi'' \vee \Box\varphi')$, respectively.

The size of an LTL formula is defined by Definition 2.1.

Definition 2.1 (size of LTL formula). Let ϕ be an LTL formula. The size of ϕ, denoted by $|\phi|$, is recursively defined as follows: if $\phi = p$, then $|\phi| = 1$; if $\phi = \mathrm{o}_1\ \phi'$, then $|\phi| = |\phi'| + 1$; if $\phi = \phi'\ \mathrm{o}_2\ \phi''$, then $|\phi| = |\phi'| + |\phi''| + 1$, where $p \in \mathbb{P} \cup \{\top\}$, $\mathrm{o}_1 \in \{\neg, \bigcirc\}$, $\mathrm{o}_2 \in \{\wedge, \mathcal{U}\}$, and ϕ', ϕ'' are LTL formulae.

The sub-formulae of an LTL formula is defined by Definition 2.2.

Definition 2.2 (sub-formulae of LTL). Let ϕ be an LTL formula. The set of sub-formulae of ϕ, denoted by $\mathsf{sub}(\phi)$, is recursively defined as follows: if $\phi = p$, then $\mathsf{sub}(\phi) = \{p\}$; if $\phi = \mathrm{o}_1\ \phi'$, then $\mathsf{sub}(\phi) = \{\phi\} \cup \mathsf{sub}(\phi')$; if $\phi = \phi'\ \mathrm{o}_2\ \phi''$, then $\mathsf{sub}(\phi) = \{\phi\} \cup \mathsf{sub}(\phi') \cup \mathsf{sub}(\phi'')$, where $p \in \mathbb{P} \cup \{\top\}$, $\mathrm{o}_1 \in \{\neg, \bigcirc\}$, $\mathrm{o}_2 \in \{\wedge, \mathcal{U}\}$, ϕ', ϕ'' are LTL formulae.

LTL formulae are interpreted over *infinite traces* of propositional states. A trace is represented in the form $\pi = s_0, \ldots, s_k, (s_{k+1}, \ldots, s_m)^\omega$, where $s_t \in 2^{\mathbb{P}}$ is a state at time t and $(s_{k+1} \ldots s_m)^\omega$ is a *loop* and means that $s_{k+1} \ldots s_m$ appears in order and infinitely. For every state s_i of π and every $p \in \mathbb{P}$, p holds if $p \in s_i$ or $\neg p$ holds otherwise. The traces mentioned in this paper are infinite. π_t denotes the *sub-trace* of π beginning from the state s_t. For brevity, π_0 is denoted by π. The *satisfaction relation* $\models$ is defined as follows:

$$\begin{array}{lll}
\pi_t \models p & \text{iff} & p \in s_t \text{ or } p = \top, \\
\pi_t \models \neg\phi & \text{iff} & \pi_t \not\models \phi, \\
\pi_t \models \phi' \wedge \phi'' & \text{iff} & \pi_t \models \phi' \text{ and } \pi_t \models \phi'', \\
\pi_t \models \bigcirc\phi & \text{iff} & \pi_{t+1} \models \phi, \\
\pi_t \models \phi' \,\mathcal{U}\, \phi'' & \text{iff} & \exists k \geq t \text{ s.t. } \pi_k \models \phi'' \text{ and} \\
 & & \forall t \leq j < k, \pi_j \models \phi',
\end{array}$$

where π is a trace, $t \in \mathbb{N}$, ϕ, ϕ', ϕ'' are LTL formulae, and $p \in \mathbb{P} \cup \{\top\}$. An LTL formula ϕ is *satisfiable* if and only if there is a trace π such

that $\pi \models \phi$. An LTL formula ϕ *implies* an LTL formula ψ, denoted by $\phi \rightarrow \psi$, if and only if $\pi \models \psi$ for every trace π such that $\pi \models \phi$. Two LTL formulae ϕ and ψ are said to be logically equivalent, denoted by $\phi \equiv \psi$, if and only if $\phi \rightarrow \psi$ and $\psi \rightarrow \phi$. Checking LTL satisfiability is to check whether an LTL formula is satisfiable or not, which is PSPACE-complete [61].

2.2 Deep Neural Network

Before introducing related deep neural networks, we introduce some common notations in deep neural networks. We use lowercase bold letters to represent vectors. We use uppercase bold letters to represent matrices. Let $\mathbf{x}_i$ and $\mathbf{x}_j$ be two vectors. $[\mathbf{x}_i, \mathbf{x}_j]$ means concatenating $\mathbf{x}_i$ and $\mathbf{x}_j$. The LTL formula ϕ and the trace π can be represented as a sequence of tokens. $\phi[i]$ and $\pi[i]$ represent the i-th character of ϕ and π, respectively.

2.2.1 Transformer. Transformer [65] is one of the most powerful sequence models. It is a neural network with an encoder-decoder architecture. The input vectors are first passed through the encoder to obtain the encoded representation vectors; then the encoded representation vectors is passed through the decoder to obtain the output vectors. Note that each time for decoding, the Transformer also needs to consider the output of the previous decoding. On the basic encoder-decoder architecture, the Transformer equips with the multi-head self-attention mechanism (Equation (1)), providing powerful feature extraction capabilities.

$$\begin{aligned} MultiHead(\mathbf{Q},\mathbf{K},\mathbf{V}) &= Concat(\mathbf{head}_1,\ldots,\mathbf{head}_h),\\ \mathbf{head}_i &= Attention(\mathbf{Q}\mathbf{W}_i^Q,\mathbf{K}\mathbf{W}_i^K,\mathbf{V}\mathbf{W}_i^V),\\ Attention(\mathbf{Q},\mathbf{K},\mathbf{V}) &= softmax(\frac{\mathbf{Q}\mathbf{K}^\top}{\sqrt{d_k}})\mathbf{V}, \end{aligned} \tag{1}$$

where $\mathbf{Q}$, $\mathbf{K}$, $\mathbf{V}$ represent the query, key, and value vector respectively, $\mathbf{head}_i$ is one of the multi-heads. $Attention(\mathbf{X},\mathbf{X},\mathbf{X})$ is the self-attention mechanism. Let $\mathbf{X} = [\mathbf{x}_1,\ldots,\mathbf{x}_n]$ be a sequence, where $\mathbf{x}_i$ is the i-th element. Intuitively, multi-head self-attention mechanism extracts features by building relationships among all $\mathbf{x}_i$ in $\mathbf{X}$.

2.2.2 Relational Graph Convolutional Network (R-GCN). R-GCN [52] is a variant of GNN. R-GCN consists of a stack of neural network layers, where each layer aggregates local neighborhood information, *i.e.*, the features of its neighbors, and then combines its own features to generate the features of the next layer. R-GCN introduces the concept of different types of edges on the graph. Based on different types of edges, R-GCN iteratively updates the embedding of vertex v, as shown in Equation (2).

$$\mathbf{x}_v^{(t)} = \sigma\left(\sum_{r\in R}\sum_{u\in \mathsf{N}(v,r)}\frac{1}{|\mathsf{N}(v,r)|}\mathbf{W}_r^{(t)}\mathbf{x}_u^{(t-1)}\right), \tag{2}$$

where R is the set of edge types, $\mathsf{N}(v,r)$ represents the set of vertices that point to the vertex v through the edge of type r, $\mathbf{W}_r^{(t)}$ are trainable parameters.

2.2.3 Recursive Neural Network (TreeNN). TreeNN [62, 63] is a general framework for encoding tree structures. Given a tree, TreeNN learns an embedding for each vertex in the tree by recursively combining the representations of its subtrees using a neural network. Algorithm 1 shows the framework of TreeNN, where τ_v represents the type of vertex v. The combination function (Combine) tends to have different definitions depending on the type of vertex. TreeNN offers two instantiations, *i.e.*, the implementation of One-hotEmbedding and Combine.

Algorithm 1: `TreeNN`

Input : Current vertex v.
Output : The embedding $\mathbf{r}_v$ of v.

```
if v is leaf then
    r_v ← One-hotEmbedding v
else
    get a set of child vertices C_v of v
    r_v ← CombineTreeNN(c_0),...,TreeNN(c_k),τ_v, where
      c_i ∈ C_v
return r_v
```

3 PRACTICAL APPLICATION

Our work is a practice in obtaining highly confident results for LTL satisfiability checking in polynomial time. This makes it possible to apply neural networks to some tasks.

One such application is for LTL-SAT-heavy tasks, *e.g.*, goal-conflict identification [16, 39]. Its purpose is to identify BCs in the form of LTL. A valid BC fulfills many satisfiability constraints which can be verified by multiple calls to LTL satisfiability checking [3]. Neural networks can act as a highly confident and efficient pre-identifier. Specifically, we first use the neural network to verify the satisfiability conditions of a candidate one by one. If the possibility of satisfiability is lower than a certain threshold, we can directly filter out these obviously invalid candidates. Then, we exploit the logical approaches to verify the more likely candidates with high possibility of satisfiability. The verification process can increase the number of traversed candidate formulae because of polynomial running time for one candidate. It also can pre-filter out obviously invalid candidates to significantly reduce the number of calls to LTL satisfiability solvers. Besides, it can guarantee soundness because logical approaches used in the post-processing.

The second application is to extract knowledge from a highly accurate and confident neural network for LTL satisfiability checking using some explanation approaches to neural networks, such as [25, 41, 49, 72]. The knowledge can be a concise satisfiable condition, which can guide the practice of LTL satisfiability checking for researchers. For example, the unsatisfiability of $\Box(p \rightarrow (\bigcirc q \wedge \bigcirc\bigcirc q \wedge \bigcirc\bigcirc\bigcirc q)) \wedge \Box(q \rightarrow (\bigcirc\neg q)) \wedge \Diamond p$ is due to a part of the formula, *i.e.*, $\Box(p \rightarrow (\bigcirc q \wedge \bigcirc\bigcirc q)) \wedge \Box(q \rightarrow (\bigcirc\neg q)) \wedge \Diamond p$. As can be seen from the example, this application can also facilitate the extraction of unsatisfiable core for LTL [54, 55].

The third application is to combine the problem of satisfiability checking and satisfiable trace generation [26]. While checking satisfiability, the combined approach can also give a satisfiable trace as proof if the formula is satisfiable. We can efficiently verify the soundness of satisfiability checking by path checking [43]. In this way, the end-to-end neural network can be used as an efficient SAT-verifiable satisfiability checker.

4 RELATED WORK

4.1 Logical Approaches to Checking LTL Satisfiability

Basic approaches to checking LTL satisfiability are based on model checking. Checking satisfiability of an LTL formula ϕ can be reduced to model checking by checking whether the universal model satisfies the negation of ϕ [50, 51]. These approaches can make full use of the latest technology of model checking, such as bounded model checking [6, 14, 15], interpolation [44], property directed reachability [8, 18]. However, LTL satisfiability checking is merely a special case of model checking where the input model is universal. There may be some performance problem for solving a special case using an approach to solving the general problem.

Some specialized LTL satisfiability checkers [5, 29, 57, 66] are based on tableau, towards constructing a model to verify the satisfiability of LTL formulae. Fisher et al. (2001) [21] examined how clausal resolution can be applied to LTL (temporal resolution) by translating arbitrary LTL formulae into the normal form. Based on the temporal resolution, Schuppan (2009) [53] investigated the notion of unsatisfiable cores of LTL. Wulf et al. (2008) [67] proposed new efficient algorithms for both LTL satisfiability checking and model checking. They work directly with alternating automata using efficient exploration techniques based on antichains. Li et al. (2013) [35] reduces LTL satisfiability checking to reachability analysis of a transition system. Their approach works on-the-fly by inspecting the formula directly, thus enabling finding a satisfying model quickly without constructing the full automaton. Further, they have proposed a series of SAT-based techniques to improve the search efficiency on the transition system and achieve the SOTA performance. Li et al. (2014) [34] presented Aalta, a new LTL satisfiability checker over both infinite and finite traces, which leveraged the power of modern SAT solvers. Li et al. (2015) and Li et al. (2019) [37, 38] improved Aalta by a new explicit reasoning framework for LTL, which significantly outperformed all existing LTL satisfiability solvers. Li et al. (2018a) [32] proposed a framework that is based on the variant of the obligation-set method and accelerated by leveraging the SOTA SAT techniques.

Although much work has been proposed to check LTL satisfiability checking, given the inherent intractability of the problem, developing new approaches over other paradigms still remains to be an interesting and necessary research direction.

4.2 Neural Networks for Logical Reasoning

There have been some studies on exploiting neural networks to logical reasoning, mainly based on capturing the logical properties.

Permutation invariance is a typical property in logical reasoning. For example, the input of SAT is in conjunctive normal form (CNF). Permuting the positions of any two variables or clauses, and the result is unchanged. Zaheer et al. (2017) [70] designed a general architecture to capture permutation invariance. They pointed out that designing permutation-invariant functions to aggregate elemental features in sets is the key. Simple permutation invariant functions include summation function, maximum function, average function, *etc.*. Lee et al. (2019) [31] proposed Set Transformer, an extension of Transformer, to represent sets. They exploited an attention mechanism to design permutation-invariant functions. The advantage of attention is in its strong ability to express individual characteristics of the elements of a collection. But it has a high computational complexity. Cameron et al. (2020) [11] studied the pros and cons of the exchangeable matrix architecture [28] and GNN, based on the message passing mechanism [59] in solving random 3-SAT. Their conclusion is that the permutation invariance expression ability of the two methods is comparable, but the generalization ability of the exchangeable matrix architecture is slightly stronger than that of the GNN. Other work [45, 73] proposed to exploit a special pooling technique as a permutation-invariant function.

Some logical representations have tree structure, *e.g.*, non-clausal Boolean formulae [40] and higher-order logical formulae [46]. The embedding of a tree often requires to express recursive induction, because trees and sub-trees often have recursive relationships. Allamanis et al. (2017) [2] proposed the neural equivalence network to study the problem of semantic equivalence classification of non-clausal Boolean formulae. It is based on TreeNN to represent the syntactic tree of a formula in a high-dimensional space. The embeddings of semantically equivalent formulae are required to be as close as possible. Evans et al. (2018) [19] further studied the potential of neural networks in checking the implication relationship between two non-clausal Boolean formulae. They designed a new formula embedding model named PossibleWorldNet.

There are also some studies utilizing GNN for embedding logical representations. They design a message passing mechanism for vertices in a graph to simulate the logical reasoning. Selsam et al. (2019) [59] modeled the input formulae of SAT as a graph structure to express the relationship among clauses, positive literals and negative literals. Such a structure has become a classic structure for modeling formulae in CNF, widely used in SAT solving models [4, 9, 11, 58, 71]. Cai et al. (2017) [10] suggested a TreeNN-like framework to enhance neural networks for program synthesis. They demonstrated excellent generalization and interpretability using a small amount of training data. Besides, their evaluation shows that in order for neural networks to learn program semantics robustly, it is necessary to incorporate some concepts from logical reasoning such as recursion.

The above studies show that neural networks are expected to capture logical semantics through training. Inspired by them, we design end-to-end neural networks that can capture as many logical semantics for checking LTL satisfiability as possible.

4.3 Embedding LTL Formulae

There are a few studies on embedding of LTL formulae. Hahn et al. (2021) [26] studied whether neural networks can capture the semantics of LTL. However, their work focuses on whether neural networks can generate its satisfiable trace given a satisfiable formula, instead of focusing on whether neural networks can check the satisfiability, which is the focus of our work. In other words, their approach does not classify the formula, but generates trace, regardless of whether the formula is satisfiable or not. Obviously, if the input formula is not satisfiable, the output trace is meaningless. Vaezipoor et al. (2021) [64] used R-GCN to embed commands in LTL to train an agent to make command-compliant decisions. Xie

et al. (2021) [68] utilized GNN for information aggregation over a deterministic finite-state automata (DFA) which is semantically equivalent to LTL.

5 MODEL ARCHITECTURE

In this section, we first introduce three logical properties of LTL, then present three classes of neural networks which are able to embed a given LTL formula, and finally describe how to use the embedding for binary classification (satisfiable or unsatisfiable).

5.1 Logical Properties of LTL

The recursive property in LTL formulae is obviously derived from the definition of the LTL syntax. Proposition 5.1 shows the recursive property of semantics. Intuitively, the satisfaction relation between a sub-trace π_i and an LTL formula ϕ recursively depends on the satisfaction relation between the sub-trace (π_i or π_{i+1}) of π_i and the sub-formula (ϕ, ϕ', or ϕ'') of ϕ.

Proposition 5.1. *Let ϕ be an LTL formula and π a trace. For any sub-trace π_i of π, it fulfills following property:*

- *if $\phi = p$, then $\pi_i \models \phi$ if and only if $\pi_i \models p$;*
- *if $\phi = \neg\phi'$, then $\pi_i \models \phi$ if and only if $\pi_i \not\models \phi'$;*
- *if $\phi = \bigcirc\phi'$, then $\pi_i \models \phi$ if and only if $\pi_{i+1} \models \phi'$;*
- *if $\phi = \phi' \wedge \phi''$, then $\pi_i \models \phi$ if and only if $\pi_i \models \phi'$ and $\pi_i \models \phi''$;*
- *if $\phi = \phi' \mathcal{U} \phi''$, then $\pi_i \models \phi$ if and only if it fulfills either the condition where $\pi_i \models \phi''$ or the condition where $\pi_i \models \phi'$ and $\pi_{i+1} \models \phi$,*

where ϕ, ϕ', ϕ'' are LTL formulae, and $p \in \mathbb{P} \cup \{\top\}$.

Proposition 5.2 and Proposition 5.3 demonstrate the permutation invariance of operators and atomic propositions, respectively. Intuitively, the permutation invariance shows that by exchanging the sub-formulae connected by the $\wedge$ operator, the satisfiability of the formula keeps.

Proposition 5.2. *Let ϕ be an LTL formula in the form of $\phi_i \wedge \phi_j$. $\phi \equiv \phi_j \wedge \phi_i$ holds.*

Proposition 5.3. *Let $\mathbb{P}$ be a set of atomic propositions and ϕ an LTL formula over $\mathbb{P}$. For any $p, q \in \mathbb{P} \cup \{\top\}$, if ϕ is satisfiable, then $\phi[p/\diamond][q/p][\diamond/q]$ is satisfiable, where $\diamond \notin \mathbb{P} \cup \{\top\}$ and $\varphi[\varphi_j/\varphi_i]$ means replacing the sub-formula φ_j of φ with φ_i.*

Proposition 5.4 demonstrates the sequentiality of LTL. Intuitively, the sequentiality shows that by exchanging two sub-formulae connected by the $\mathcal{U}$ operator, the satisfiability may change.

Proposition 5.4. *Let ϕ be an LTL formula in the form of $\phi_i \mathcal{U} \phi_j$. $\phi \equiv \phi_j \mathcal{U} \phi_i$ if and only if $\phi_i \equiv \phi_j$.*

We use Example 5.5 to illustrate the above properties.

Example 5.5. Let $\{p, q, r\}$ be a set of atomic propositions. One of the reasons $(p \wedge q) \mathcal{U} r$ is satisfiable is that we can construct a trace $\pi = \{p, q\}, (\{r\})^\omega$ such that $\pi \models p \wedge q$ and $\pi_1 \models (p \wedge q) \mathcal{U} r$ where $\pi_1 \models (p \wedge q) \mathcal{U} r$ holds because $\pi_1 \models r$. It shows the recursively defined semantics of LTL. Since $p \wedge q \equiv q \wedge p$, $(p \wedge q) \mathcal{U} r \equiv (q \wedge p) \mathcal{U} r$, which shows the permutation invariance of sub-formulae. Both $(p \wedge r) \mathcal{U} q$ and $(p \wedge q) \mathcal{U} r$ are satisfiable, which shows the permutation invariance of atomic propositions. $(r \mathcal{U} q) \wedge \Box\neg r$ is satisfiable while $(q \mathcal{U} r) \wedge \Box\neg r$ is unsatisfiable. The semantic inequality of $r \mathcal{U} q$ and $q \mathcal{U} r$ leads to different satisfiability results, which reflects the sequentiality.

5.2 Embedding Based on Transformer

An LTL formula can be treated as a sequence of tokens. The simplest embedding approach is to apply a sequence model, such as Transformer [65], to the input formula. In practice, Hahn et al. (2021) [26] shows that it is possible to train a Transformer to generate LTL satisfiable traces. Therefore, we explore whether Transformer can check the satisfiability of LTL.

We modify the model of [26] and denote it as `Transformer`. We use the Transformer encoder to embed a prefixed expression of LTL as the following form:

$$[CLS]\ \phi[0]\ \phi[1] \cdots \phi[|\phi| - 1]\ [EOS],$$

where "[*CLS*]" and "[*EOS*]" are two special tokens that represent the start and end of the sequence of tokens, respectively, ϕ is an LTL formula in prefixed expression, $\phi[i]$ is the i-th token of ϕ. Neural networks need to distinguish atomic propositions in high-dimensional space and the atomic propositions are only symbols, so we use one-hot vectors $\mathbf{x}_p \in \mathbb{R}^{d_m}$ as the initial embedding for atomic propositions and set them to be non-trainable, where d_m is a hyperparameter. Besides, we use trainable vectors $\mathbf{x}_\top, \mathbf{x}_{op}, \mathbf{x}_{[CLS]}, \mathbf{x}_{[EOS]} \in \mathbb{R}^{d_m}$ to represent $\top$, logical operators, "[*CLS*]", and "[*EOS*]", respectively.

Since LTL satisfiability checking is a kind of binary classification, instead of using the decoder of Transformer, we apply a one-layer multi-layer perceptron (MLP) with softmax activation to transform the embedding of "[*CLS*]" to a 2-dimensional vector indicating the binary result. Similar processing is widely used in the field of natural language processing to extract features from input sentences (a sequence of tokens). We train `Transformer` by minimizing the cross-entropy loss between $\hat{y_\phi}$ and the ground truth y_ϕ, where if ϕ is satisfiable, then $y_\phi = 1$; otherwise, $y_\phi = 0$. We use Example 5.6 to show how to embed an LTL formula by `Transformer`.

Example 5.6. Let $\mathbb{P} = \{p, q, r\}$ be a set of atomic propositions and $\phi = (p \wedge q)\ \mathcal{U}\ \bigcirc r$ an LTL formula. In order to embed atomic propositions by the one-hot encoding, we set d_m to 3. Suppose the embeddings of atomic propositions are $\mathbf{x}_p = [1, 0, 0]^\top$, $\mathbf{x}_q = [0, 1, 0]^\top$, and $\mathbf{x}_r = [0, 0, 1]^\top$ and the embeddings of other tokens are trainable denoted by $\mathbf{x}_{[CLS]}$, $\mathbf{x}_{[EOS]}$, $\mathbf{x}_\mathcal{U}$, $\mathbf{x}_\wedge$, and $\mathbf{x}_\bigcirc$. Because of the prefixed expression of ϕ, the input of `Transformer` is $\mathbf{X}_\phi = \left[\mathbf{x}_{[CLS]}, \mathbf{x}_\mathcal{U}, \mathbf{x}_\wedge, \mathbf{x}_p, \mathbf{x}_q, \mathbf{x}_\bigcirc, \mathbf{x}_r, \mathbf{x}_{[EOS]}\right]$. We compute the new embedding of "[*CLS*]" by $Attention(\mathbf{X}_\phi, \mathbf{X}_\phi, \mathbf{X}_\phi)$ in Equation (1) and apply a one-layer MLP with softmax activation to it to obtain $\hat{y_\phi}$.

5.3 Embedding Based on GNN

While sequential models can learn the permutation invariance and sequentiality, given the recursively defined syntactics and semantics of LTL, GNN may provide better inductive bias. This choice is consistent with recent work in programming languages and verification, where GNN is used to embed the abstract syntax tree of input programs [1, 60]. Recently, Vaezipoor et al. (2021) [64] embedded commands expressed in LTL formulae using relational graph convolutional network (R-GCN). Therefore, we naturally wanted

to explore whether R-GCN can check LTL satisfiability. To this end, we modify the model [64] to check LTL satisfiability, denoted by RGCN. For a given LTL formula ϕ, we embed ϕ through its labeled graph $G_\phi^L = \left(V_\phi^L, E_\phi^L, R_\phi^L, \mathtt{lab}_\phi^L\right)$ formally defined below.

Definition 5.7 (labeled graph of LTL [64]). Let $\mathbb{P}$ be a set of atomic propositions and ϕ a LTL formula over $\mathbb{P}$. The labeled graph of ϕ is tuple $G_\phi^L = \left(V_\phi^L, E_\phi^L, R_\phi^L, \mathtt{lab}_\phi^L\right)$ defined as follows. V_ϕ^L is a set of vertices. Specially, $r_\phi \in V_\phi$ is root vertex. $\mathtt{lab}_\phi^L \colon V_\phi^L \to \{\top\} \cup \mathbb{P} \cup \{\neg, \wedge, \bigcirc, \mathcal{U}\}$ is a labeled function. $R_\phi^L = \{SL, UN, BL, BR\}$ is a set of edge types, where SL indicates a self-loop from a vertex to itself, UN indicates the edge pointing to the vertex labeled as a unary operator, BL indicates the edge from a left child vertex to the vertex labeled as a binocular operator, and BR indicates the edge from a right child vertex to the vertex labeled as a binocular operator. $E_\phi^L \subseteq V_\phi^L \times R_\phi^L \times V_\phi^L$ is a set of directed edges. V_ϕ^L and E_ϕ^L are initialized as $\{v_\phi\}$ and $\emptyset$, respectively. We set $r_\phi = v_\phi$. $\phi_i \in \mathsf{sub}(\phi)$, V_ϕ^L, E_ϕ^L, and $\mathtt{lab}_\phi^L$ are computed as follows:

- if $\phi_i = p$, then $E_\phi^L = E_\phi^L \cup \{\langle v_{\phi_i}, SL, v_{\phi_i}\rangle\}$ and $\mathtt{lab}_\phi^L(\phi_i) = p$;
- if $\phi_i = \mathrm{o}_1\ \phi_j$, then $V_\phi^L = V_\phi^L \cup \{v_{\phi_j}\}$, $E_\phi^L = E_\phi^L \cup \{\langle v_{\phi_j}, UN, v_{\phi_i}\rangle, \langle v_{\phi_j}, SL, v_{\phi_i}\rangle, \langle v_{\phi_i}, SL, v_{\phi_i}\rangle\}$, $\mathtt{lab}_\phi^L(\phi_i) = \mathrm{o}_1$;
- if $\phi_i = \phi_j\ \mathrm{o}_2\ \phi_k$, then $V_\phi^L = V_\phi^L \cup \{v_{\phi_j}, v_{\phi_k}\}$, $E_\phi^L = E_\phi^L \cup \{\langle v_{\phi_j}, BL, v_{\phi_i}\rangle, \langle v_{\phi_k}, BR, v_{\phi_i}\rangle, \langle v_{\phi_i}, SL, v_{\phi_i}\rangle\}$, $\mathtt{lab}_\phi^L(\phi_i) = \mathrm{o}_2$,

where $p \in \mathbb{P} \cup \{\top\}$, $\mathrm{o}_1 \in \{\neg, \bigcirc\}$, $\mathrm{o}_2 \in \{\wedge, \mathcal{U}\}$, and ϕ_j, ϕ_k are LTL formulae.

Figure 1 illustrates an example of labeled graph.

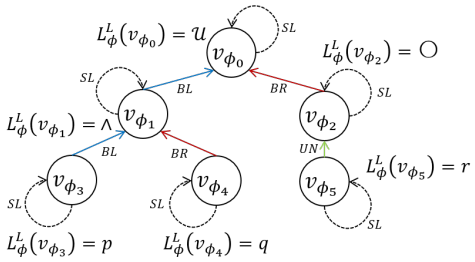


Figure 1: The labeled graph of $\phi = (p \wedge q)\ \mathcal{U}\ \bigcirc r$. $V_\phi^L = \{v_{\phi_0}, v_{\phi_1}, v_{\phi_2}, v_{\phi_3}, v_{\phi_4}, v_{\phi_5}\}$ and $r_\phi = v_{\phi_0}$. Edges marked by dashed lines are of type SL, edges marked by green are of type UN, edges marked by blue are of type BL, and edges marked by red are of type BR.

In addition, we add a global vertex and for every vertex v in the labeled graph, add an edge labeled by the new type GV from v pointing to the global vertex. We initialize vectors for vertices in the labeled graph and the global vertex. Specifically, we use one-hot vectors $\mathbf{x}_p \in \mathbb{R}^{d_m}$ as the initial embedding of the vertices labeled by atomic propositions and use the trainable vectors $\mathbf{x}_\top, \mathbf{x}_{op}, \mathbf{x}_g \in \mathbb{R}^{d_m}$ to initialize the vertices labeled by $\top$, logical operators, and the global vertex, respectively. We iteratively update these vectors based on the type of the edges through message passings. At iteration $t+1$, the embedding $\mathbf{x}_v^{(t+1)} \in \mathbb{R}^{d_m}$ of vertex v is updated by Equation (3).

$$\mathbf{x}_v^{(t+1)} = \sigma\left(\sum_{r \in R_\phi^L \cup \{GV\}} \sum_{u \in \mathsf{N}(v,r)} \frac{1}{|\mathsf{N}(v,r)|} \mathbf{W}_r \mathbf{x}_u^{(t)}\right), \tag{3}$$

where $\mathsf{N}(v,r)$ represents the set of vertices that point to vertex v through an edge of type r and σ is the ReLU activation function. We set a trainable $\mathbf{W}_r$ for each $r \in R_\phi^L \cup \{GV\}$ and share $\mathbf{W}_r$ only for edges of the same type per iteration.

After $T \in \mathbb{N}$ iterations, we treat the embedding $\mathbf{x}_g^{(T)}$ of the global vertex as the embedding of ϕ. Then, we apply a one-layer MLP with softmax activation to $\mathbf{x}_g^{(T)}$ to obtain the classification probability $\hat{y_\phi}$. We train RGCN to minimize the cross-entropy loss between $\hat{y_\phi}$ and the ground truth y_ϕ.

5.4 Embedding Based on TreeNN

To capture the recursive property of LTL, the natural idea is to adopt the general framework – TreeNN [62, 63] to embed the syntax tree of LTL formulae. The syntax tree of LTL is defined below whereas Figure 2 shows an example of it.

Definition 5.8 (syntax tree of LTL). Let ϕ be an LTL formula. The syntax tree T_ϕ of ϕ is a tuple $\left(V_\phi, E_\phi, r_\phi, \mathtt{lab}_\phi\right)$ defined as follows. V_ϕ is a set of vertices. $r_\phi \in V_\phi$ is the root vertex. $E_\phi \subseteq V_\phi \times V_\phi$ is a set of undirected edges. $\mathtt{lab}_\phi \colon V_\phi \to \{\top\} \cup \mathbb{P} \cup \{\neg, \wedge, \bigcirc, \mathcal{U}\}$ is a labeled function. V_ϕ and E_ϕ are initialized as $\{v_\phi\}$ and $\emptyset$. We set $r_\phi = v_\phi$. $\phi_i \in \mathsf{sub}(\phi)$, V_ϕ, E_ϕ, and $\mathtt{lab}_\phi$ are computed as follows:

- if $\phi_i = p$, then $\mathtt{lab}_\phi(\phi_i) = p$;
- if $\phi_i = \mathrm{o}_1\ \phi_j$, then $V_\phi = V_\phi \cup \{v_{\phi_j}\}$, $E_\phi = E_\phi \cup \{(v_{\phi_i}, v_{\phi_j})\}$, $\mathtt{lab}_\phi(\phi_i) = \mathrm{o}_1$;
- if $\phi_i = \phi_j\ \mathrm{o}_2\ \phi_k$, then $V_\phi = V_\phi \cup \{v_{\phi_j}, v_{\phi_k}\}$, $E_\phi = E_\phi \cup \{(v_{\phi_i}, v_{\phi_j}), (v_{\phi_i}, v_{\phi_k})\}$, $\mathtt{lab}_\phi(\phi_i) = \mathrm{o}_2$,

where, $p \in \mathbb{P} \cup \{\top\}$, $\mathrm{o}_1 \in \{\neg, \bigcirc\}$, $\mathrm{o}_2 \in \{\wedge, \mathcal{U}\}$, and ϕ_j, ϕ_k are LTL formulae.

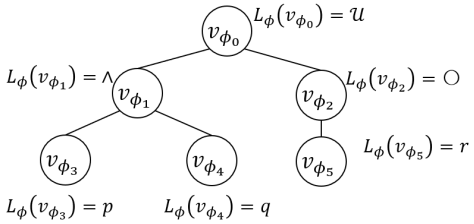


Figure 2: The syntax tree of $\phi = (p \wedge q)\ \mathcal{U}\ \bigcirc r$, where $V_\phi = \{v_{\phi_0}, v_{\phi_1}, v_{\phi_2}, v_{\phi_3}, v_{\phi_4}, v_{\phi_5}\}$ and $r_\phi = v_0$.

Algorithm 2 shows the pseudo-code for embedding LTL formulae based on TreeNN. We learn the embedding of a sub-formula for

each vertex in a syntax tree by recursively aggregating and combining the embeddings of the sub-formulae using neural networks. We use a non-trainable one-hot vector $\mathbf{v}_p \in \mathbb{R}^{d_m}$ to represent an atomic proposition and use a trainable one-hot vector $\mathbf{v}_\top \in \mathbb{R}^{d_m}$ to represent $\top$. If the sub-formula is an atomic proposition or $\top$, denoted by p, we project the $\mathbf{v}_p \in \mathbb{R}^{d_m}$ to $\mathbf{r}_p \in \mathbb{R}^{d_h}$ by a trainable projection matrix, where d_h is a hyperparameter (line 2 of Algorithm 2). Otherwise, we perform a combination function (Combine) with different trainable parameters for different logical operators (line 6 and 12 of Algorithm 2). Algorithm 3 shows the pseudo-code for Combine. We use a two-layer MLP with a residual-like connection to compute the combined embedding of multiple sub-formulae, which is the same as in the work [2]. This way can enable more flexibility in modeling the semantics of LTL mapping of syntax trees. The dimensions of $\mathbf{W}_{0,op}, \mathbf{W}_{1,op}, \mathbf{W}_{2,op}$ are different, which is related to the input embedding dimension of the operator.

Algorithm 2: LTLTreeNN

Input : A syntax tree $\left(V_\phi, E_\phi, r_\phi, \mathtt{lab}_\phi\right)$ of ϕ and a current vertex r_{ϕ_i}.
Output : The embedding $\mathbf{r}_{\phi_i}$ of ϕ_i.

1 **if** $\mathtt{lab}_\phi\left(r_{\phi_i}\right) = p$, *where* $p \in \mathbb{P} \cup \{\top\}$ **then**
2 $\mathbf{r}_{\phi_i} \leftarrow$ One-hotEmbedding$p$
3 **else if** $\mathtt{lab}_\phi\left(r_{\phi_i}\right) = op_1$, *where* $op_1 \in \{\neg, \bigcirc\}$ **then**
4 get $r_{\phi_j} \in V_\phi$ such that $\left(r_{\phi_i}, r_{\phi_j}\right) \in E_\phi$
5 $\mathbf{r}_{\phi_j} \leftarrow$ LTLTreeNN$\left(V_\phi, E_\phi, r_\phi, \mathtt{lab}_\phi\right)$,$r_{\phi_j}$
6 $\mathbf{r}_{\phi_i} \leftarrow$ Combine$\mathbf{r}_{\phi_j}$,op_1
7 **else** /* $\mathtt{lab}_\phi\left(r_\phi\right) \in \{\wedge, \mathcal{U}\}$ */
8 get $r_{\phi_j}, r_{\phi_k} \in V_\phi$ such that $\left(r_{\phi_i}, r_{\phi_j}\right), \left(r_{\phi_i}, r_{\phi_k}\right) \in E_\phi$
9 $\mathbf{r}_{\phi_j} \leftarrow$ LTLTreeNN$\left(V_\phi, E_\phi, r_\phi, \mathtt{lab}_\phi\right)$,$r_{\phi_j}$
10 $\mathbf{r}_{\phi_k} \leftarrow$ LTLTreeNN$\left(V_\phi, E_\phi, r_\phi, \mathtt{lab}_\phi\right)$,$r_{\phi_k}$
11 $\mathbf{r}_{\phi_j,\phi_k} \leftarrow$ Aggregate$\mathbf{r}_{\phi_j}$,$\mathbf{r}_{\phi_k}$
12 $\mathbf{r}_{\phi_i} \leftarrow$ Combine$\mathbf{r}_{\phi_j,\phi_k}$,$\mathtt{lab}_\phi\left(r_\phi\right)$
13 **return** $\mathbf{r}_\phi$

Algorithm 3: Combine

Input : An aggregation $\mathbf{r}$ of the embeddings of sub-formulae and a logical operator op.
Output : An embedding $\mathbf{r}_{out}$.

1 $\mathbf{r}' \leftarrow \sigma\left(\mathbf{W}_{0,op} \cdot \mathbf{r}\right)$ /* σ is the ReLU activation function. */
2 $\mathbf{r}_{out} \leftarrow \mathbf{W}_{1,op} \cdot \mathbf{r}' + \mathbf{W}_{2,op} \cdot \mathbf{r}$
3 **return** $\mathbf{r}_{out} / \|\mathbf{r}_{out}\|_2$

Different aggregation functions lead to different variants of TreeNN. We consider two basic aggregation functions as follows:

- we aggregate the embeddings of multiple sub-formulae by performing an MP, denoted by `TreeNN-MP`;
- we aggregate the embeddings of multiple sub-formulae by concatenating the embeddings of the sub-formulae in order, denoted by `TreeNN-con`.

Algorithm 4: SubexpAe [2]

Input : A formula embedding $\mathbf{r}_\phi$ of an LTL formula ϕ, the embedding $\mathbf{r}_c$ by concatenating the embeddings of all sub-formulae of ϕ, and a logical operator op, where $\phi = \phi_i \; op \; \phi_j$ or $\phi = op \; \phi_i$.
Output : A loss value.

1 $\tilde{\mathbf{r}_c} \leftarrow \tanh\left(\mathbf{W}_d \cdot \tanh\left(\mathbf{W}_{e,op} \cdot \left[\mathbf{r}_\phi, \mathbf{r}_c\right] \cdot \mathbf{n}\right)\right)$
2 $\tilde{\mathbf{r}_c} \leftarrow \tilde{\mathbf{r}_c} \cdot \|\mathbf{r}_c\|_2 / \|\tilde{\mathbf{r}_c}\|_2$
3 $\tilde{\mathbf{r}_\phi} \leftarrow$ Combine$\tilde{\mathbf{r}_c}$,op
4 **return** $-\left(\tilde{\mathbf{r}_c}^{\mathrm{T}} \mathbf{r}_c + \tilde{\mathbf{r}_\phi}^{\mathrm{T}} \mathbf{r}_\phi\right)$

For a given LTL formula ϕ, we compute the probability $\hat{y_\phi}$ by applying a one-layer MLP with softmax activation to $\mathbf{r}_\phi$ defined below.

$$\mathbf{r}_\phi = \text{LTLTreeNN}\left(\left(V_\phi, E_\phi, r_\phi, \mathtt{lab}_\phi\right), \phi\right) \tag{4}$$

We train the variants of TreeNN to minimize the cross-entropy loss between $\hat{y_\phi}$ and the ground truth y_ϕ.

We also study two variants of TreeNN.

The first is EQNET [2]. It adds a regularization in order to encourage the embeddings to become more close to the semantics, reducing their dependence on surface-level syntactic information. This method also applies to LTL satisfiability checking, since the satisfiability of LTL formulae does not depend on their syntax. We follow [2] to modify EQNET to check LTL satisfiability, denoted by `EQNET`. We use Algorithm 2 and the concatenation function to embed LTL formulae, like `TreeNN-con`, but `EQNET` adds a new loss function (Algorithm 4) in training. Given any sub-formulae ϕ, Algorithm 4 aims to auto-encode the representation tuple $\left[\mathbf{r}_\phi, \mathbf{r}_c\right]$ into a low-dimensional space with a de-noising auto-encoder (line 1 of Algorithm 4), where $\mathbf{n}$ is a binary noise vector with k percent of elements setting to zero and $\mathbf{W}_d, \mathbf{W}_{e,op}$ are trainable parameters. Then, Algorithm 4 computes the reconstruction error (line 4 of Algorithm 4). The motivation behind this is to remove irrelevant information from the representations by auto-encoding the formula and its sub-formulae representations together. We train `EQNET` to minimize not only the cross-entropy loss between $\hat{y_\phi}$ and the ground truth y_ϕ, but also the mean value of SubexpAe for all the sub-formulae of the input formula.

The second variant is `TreeNN-inv`, which is based on `TreeNN-con` and adds a new loss function defined by Equation (5) in training.

$$\begin{aligned} L_{inv}(\phi) &= \sum_{\phi_i = \phi_j \wedge \phi_k \in \mathsf{sub}(\phi)} \left(1 - \mathrm{CS}(\mathbf{r}_{\phi_j,\phi_k}, \mathbf{r}_{\phi_k,\phi_j})\right), \\ \mathbf{r}_{\phi_j,\phi_k} &= \text{Combine}([\mathbf{r}_{\phi_j}, \mathbf{r}_{\phi_k}], \wedge), \\ \mathbf{r}_{\phi_k,\phi_j} &= \text{Combine}([\mathbf{r}_{\phi_k}, \mathbf{r}_{\phi_j}], \wedge), \\ \mathrm{CS}(\mathbf{x}_1, \mathbf{x}_2) &= \frac{\mathbf{x}_1 \cdot \mathbf{x}_2}{\|\mathbf{x}_1\| \, \|\mathbf{x}_2\|}, \end{aligned} \tag{5}$$

where $\mathbf{x}_1$ and $\mathbf{x}_2$ are vectors. The primary intuition behind Equation (5) is that the embedding of $\phi_j \wedge \phi_k$ and that of $\phi_k \wedge \phi_j$ require

to be as close as possible to keep the permutation invariance. We use the cosine distance to measure the similarity between two embeddings. Similar to EQNET, we train TreeNN-inv to minimize not only the cross-entropy loss between $\hat{y_\phi}$ and the ground truth y_ϕ, but also $L_{inv}(\phi)$ defined in Equation (5).

5.5 Summary of Embedding Approaches

The reason for choosing Transformer and R-GCN for checking LTL satisfiability is that their usage in embedding LTL has been verified: Transformer has been used for generating LTL satisfiable trace [26] and R-GCN has been used for embedding LTL commands [64]. The reason why we compare TreeNN with Transformer and R-GCN is that its architecture is in line with the recursive property of LTL. We do not consider the model [68] because translating LTL to DFA is in 2EXPTIME [30], which leads to a severe time overhead for transforming complex formulae with more than 10^4 characters.

These three classes of neural networks have varying degrees for keeping the logical properties of LTL.

Transformer does not keep any logical properties of LTL, but it can learn to express permutation invariance and sequentiality thanks to the powerful multi-head self-attention mechanism [11]. The primary intuition behind it is that Transformer builds a relationship between each element in the sequence to deduce which elements are permutation invariant and which are sequential.

R-GCN expresses the sequentiality but cannot keep the permutation invariance. Besides, R-GCN does not distinguish semantics of different logical operators, but only the types of edges.

Proposition 5.9. *Let ϕ be an LTL formula and $\mathbf{W}_{BL}$ and $\mathbf{W}_{BR}$ two trainable parameters of a R-GCN. If $\phi_i = \phi_j$ op $\phi_k \in \mathsf{sub}(\phi)$ and $\mathbf{W}_{BL} \neq \mathbf{W}_{BR}$, then $\mathbf{x}^{(t+1)}_{v_{\phi_j \ op \ \phi_k}} = \mathbf{x}^{(t+1)}_{v_{\phi_k \ op \ \phi_j}}$ if and only if $\phi_j = \phi_k$, where op is a binary operator.*

It is straightforward to prove Proposition 5.9 because for each $\phi_i = \phi_j$ op $\phi_k \in \mathsf{sub}(\phi)$, we have $\mathbf{x}^{(t+1)}_{v_{\phi_j \ op \ \phi_k}} = \mathbf{W}_{SL}\mathbf{x}^{(t)}_{\phi_i} + \mathbf{W}_{BL}\mathbf{x}^{(t)}_{\phi_j} + \mathbf{W}_{BR}\mathbf{x}^{(t)}_{\phi_k}$ and $\mathbf{x}^{(t+1)}_{v_{\phi_k \ op \ \phi_j}} = \mathbf{W}_{SL}\mathbf{x}^{(t)}_{\phi_i} + \mathbf{W}_{BL}\mathbf{x}^{(t)}_{\phi_k} + \mathbf{W}_{BR}\mathbf{x}^{(t)}_{\phi_j}$.

Proposition 5.10. *Let ϕ be an LTL formula. If $\phi_i = \phi_j$ op $\phi_k \in \mathsf{sub}(\phi)$, then* Combine$(\mathsf{MP}(\mathbf{r}_{\phi_j}, \mathbf{r}_{\phi_k}), op) =$ Combine$(\mathsf{MP}(\mathbf{r}_{\phi_k}, \mathbf{r}_{\phi_j}), op)$, *where op is a binary operator and* MP *is a mean pooling function.*

We can straightforwardly prove Proposition 5.10 because $\mathsf{MP}(\mathbf{r}_{\phi_j}, \mathbf{r}_{\phi_k}) = \mathsf{MP}(\mathbf{r}_{\phi_k}, \mathbf{r}_{\phi_j})$. The MP function of a set of input vectors ensures that the order of vectors does not affect the output vector, which keeps the permutation invariance of LTL.

Proposition 5.11. *Let ϕ be an LTL formula. If $\phi_i = \phi_j$ op $\phi_k \in \mathsf{sub}(\phi)$, then* Combine$([\mathbf{r}_{\phi_j}, \mathbf{r}_{\phi_k}], op) =$ Combine$([\mathbf{r}_{\phi_k}, \mathbf{r}_{\phi_j}], op)$ *if and only if $\phi_j = \phi_k$, where op is a binary operator.*

Proposition 5.11 can also be proved because $[\mathbf{r}_{\phi_j}, \mathbf{r}_{\phi_k}] = [\mathbf{r}_{\phi_k}, \mathbf{r}_{\phi_j}]$ if and only if $\mathbf{r}_{\phi_j} = \mathbf{r}_{\phi_k}$. It shows that the concatenation function keeps the sequentiality of LTL.

TreeNN can capture the recursive property of LTL thanks to its recursive algorithmic structure (Algorithm 2). Because of using MP as the aggregation function, TreeNN-MP keeps the permutation invariance, but it does not keep the sequentiality. TreeNN-con, EQNET, and TreeNN-inv employ the concatenation function, so they can keep the sequentiality. Moreover, EQNET bridges semantic features to syntactic features via Algorithm 4, whereas TreeNN-inv also keeps the permutation invariance through Equation (5).

This summary enables us to analyze the impact of degrees for keeping different logical properties of LTL on the performance of checking LTL satisfiability.

6 DATA GENERATION

We automatically generate some synthetic datasets with different formula sizes to train and test neural networks. The synthetic datasets consist of random LTL formulae. We use the randltl tool from SPOT [17][1] to generate random formulae and label them (satisfiable or unsatisfiable) using nuXmv [12]. randltl generates unique formulae within a certain size range under a given symbolic ($\mathbb{P}$, $\top$, and logical operators) distribution. We set the size of the set of atomic propositions to 1024. Our symbolic distribution weights are as follows: the weight of $\mathbb{P}$ is 2.5 (the atomic propositions in $\mathbb{P}$ follow a uniform distribution) and that of $\top$, $\neg$, $\wedge$, $\bigcirc$, $\mathcal{U}$ are all 1.

The first synthetic dataset consists of the formulae whose sizes are in [100, 200), denoted by *SPOT*-[100, 200). There are 160K formulae as training set, 20K formulae as validation set, and 20K formulae as test set. Furthermore, we generate random formulae to construct extra test sets and divide them into six datasets according to their size: [200, 250), [250, 300), [300, 350), [350, 400), [400, 450), and [450, 500). Every extra test set consists of 2K formulae. For each training, validation, and test set, we balance the numbers of satisfiable and unsatisfiable formulae. Moreover, we ensure that formulae in the training, validation and test set are not repeated. Algorithm 5 summarizes the process of dataset generation.

Algorithm 5: Generating synthetic datasets.

Input : The size of dataset N.
Output : A synthetic dataset D.

```
D, D' ← ∅
while the number of satisfiable formulae or that of
  unsatisfiable formulae is less than N/2 do
    φ ← generate a random formula via randltl
    if φ is not in D' then
        st ← compute whether it is satisfiable or
          unsatisfiable via nuXmv
        D' ← D' ∪ {(φ, st)}
D ← randomly select N/2 satisfiable formulae and N/2
  unsatisfiable formulae from D
return D
```

In all synthetic datasets, the formulae size is less than 500 and the number of atomic propositions contained is no more than 1024. It ensures us to generate a large amount of data in an acceptable time to train and test neural networks. Since checking LTL satisfiability is computationally intractable (PSPACE-complete), even current SOTA approaches are too time-consuming to check long formulae with a large number of atomic propositions. Experimental results reported in Table 3 for nuXmv and Aalta on large scale datasets

[1] SPOT:https://spot.lrde.epita.fr/.

confirm this. Besides, we discover that unsatisfiable formulae randomly sampled by our method are sparse. Specifically, we sampled 50M formulae, of which only about 0.4M formulae are unsatisfiable. It is costly to sample a sufficient number of unsatisfiable formulae to maintain the balance of satisfiable and unsatisfiable examples in the datasets and to avoid repeatedly sampling.

7 EXPERIMENT

We consider the following three research questions.

RQ 1. *Can neural networks capture features for checking LTL satisfiability?*

RQ 2. *How well do neural networks generalize across formula sizes and distributions?*

RQ 3. *Are neural networks and SOTA logical approaches comparable on large scale datasets?*

7.1 Competitor

The competitors include 6 neural networks `Transformer`, `RGCN`, `TreeNN-MP`, `TreeNN-con`, `EQNET` and `TreeNN-inv`, as well as two logical approaches `nuXmv` and `Aalta`.

`nuXmv` [12]. It is one of the SOTA approaches for model checking. Since LTL satisfiability checking can be reduced to model checking [51], `nuXmv` is also an efficient approach for LTL satisfiability checking thanks to the SOTA technologies of model checking.

`Aalta` [37, 38]. It is one of the SOTA approaches for checking LTL satisfiability. Li et al. (2015) and Li et al. (2019) performed LTL satisfiability checking by reducing the problem to the reachability problem on a transition system. Therefore, they can solve the problem efficiently with the help of SAT solvers.

7.2 Dataset

We use synthetic datasets introduced in Section 6 to train and test neural networks. Since LTL shares the same syntax as linear temporal logic over finite traces (LTL$_f$) [24], we also evaluate our approaches using the large scale LTL$_f$ datasets [33][2] for more extensive experiments. The large scale datasets are also used to evaluate `nuXmv` and `Aalta` in the work [33]. An overview of them is given as follows.

LTL-as-LTL$_f$ [56]. It collects formulae from several prior work and has a total of 7442 formulae including 3721 pattern formulae and their negations. It consists of 6 different formula classes from different scenarios, namely acacia, alaska, anzu, forobots, rozier, and schuppan.

LTL$_f$-Specific. It consists of 1700 formulae generated by common LTL templates, summarized below. Random conjunction formulae, in a similar way to the work [35], are generated by randomly combining 19 common LTL formulae [23, 48]. Template formulae include 7 extensible templates inspired by the work [13].

NASA-Boeing. It contains 63 specifications coming from industries: 49 specifications for the design of the Boeing AIR 6110 wheelbraking system [7]; 14 specifications for the design of the NASA NextGen air traffic control system [22].

[2]The datasets and the experimental results reported by [33] are available at https://drive.google.com/open?id=1eOYGvm3C8sQ-9iyfZ8qx42K54hgrFNTC.

Table 1: The quantitative statistics of datasets, where "$|\phi|$" means the average formula size and "$|\mathbb{P}|$" means the average number of atomic propositions.

	SPOT						
	[100, 200)	[200, 250)	[250, 300)	[300, 350)	[350, 400)	[400, 450)	[450, 500)
$\|\phi\|$	151.09	225.58	275.20	325.18	375.43	425.05	474.44
$\|\mathbb{P}\|$	34.27	46.74	53.70	59.72	65.09	69.58	73.43

	LTL-as-LTL$_f$	*LTL$_f$-Specific*	*NASA-Boeing*	*DECLARE*
$\|\phi\|$	449.18	992.86	2,973.13	7,878.93
$\|\mathbb{P}\|$	25.86	228.74	96.40	41.81

DECLARE. It consists of a total of 112 LTL$_f$ template formulae widely used in business process management [20].

Table 1 shows some quantitative statistics of datasets. Note that the size of formulae in the four large scale datasets is usually much larger than that in synthetic datasets. Besides, the large scale datasets are created in a way different from synthetic datasets and therefore their formula distributions are different.

7.3 Setup

We train all neural networks on the training set of *SPOT*-[100, 200), and then test all neural networks on the test set of *SPOT*-[100, 200) to demonstrate their performance on the data with the similar formula size and distribution as in the training set. Besides, we test all neural networks on the *SPOT* with larger formulae to demonstrate their generalization ability across formula sizes.

We also evaluate all approaches on the large scale datasets. Specifically, we train all neural networks on synthetic datasets and then directly test them on the large scale datasets. This is to evaluate the generalization ability across formula sizes and formula distributions of neural networks, which is useful for industrial instances where the distribution is not clear and the amount of data is small. This evaluation also makes a good comparison between neural networks and the current SOTA logical approaches.

The evaluation metrics of performance include accuracy, precision, recall, and F1 score for binary classification, defined as Equation (6), where TP, TN, FP, and FN represent the number of true positive, true negative, false positive, and false negative respectively. In addition, we also compare the running time of each approach. We set a time limit of 3600 seconds for computing each test case. The running time of a timeout, out-of-memory, or running-error test case is uniformly assigned 3600 seconds.

$$\begin{aligned} \text{accuracy} &= \frac{\text{TP}+\text{TN}}{\text{TP}+\text{TN}+\text{FP}+\text{FN}}, \\ \text{precision} &= \frac{\text{TP}}{\text{TP}+\text{FP}}, \\ \text{recall} &= \frac{\text{TP}}{\text{TP}+\text{FN}}, \\ \text{F1} &= 2\times\frac{\text{precision}\times\text{recall}}{\text{precision}+\text{recall}}. \end{aligned} \tag{6}$$

Both training and testing of all neural networks are conducted on a single GPU (NVIDIA A100). We train all neural networks using the Adam optimizer. We use grid search to find optimal hyperparameters. The parameters common to all neural networks are as

Table 2: Evaluation results on the test set of *SPOT*-[100, 200), where "acc." means the accuracy (%), "pre." means the precision (%), "rec." means the recall (%), "F1" means the F1 score (%), and "time" indicates the sum of the running time for all formulae (seconds). Bold numbers mark better results.

approach	acc.	pre.	rec.	F1	time
Transformer	70.60	71.02	69.61	70.31	**57.09**
RGCN	65.42	71.06	52.01	60.06	3,642.99
TreeNN-MP	86.15%	90.55%	80.73%	85.36%	1,792.11
TreeNN-con	**93.76%**	**98.17%**	**89.19%**	**93.47%**	1,814.88
EQNET	90.73%	94.87%	86.13%	90.29%	485.08
TreeNN-inv	91.79%	96.23%	87.00%	91.38%	416.96

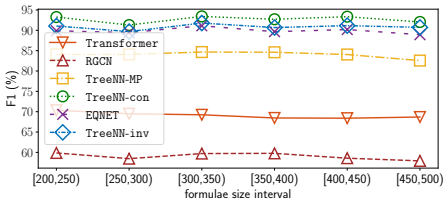


Figure 3: Evaluation results on the larger formulae. The accuracy, precision, and recall for neural networks have the same rankings and trends as F1 score.

follows: $d_m = 1024$; the maximum epoch is 256; The different parameters of Transformer are as follows: the number of network layers is 3; the dimension of the hidden layer in the encoder layer is 256; the number of self-attention heads is 4; the learning rate is $2.5e-5$; the batch size is 512. The different parameters of RGCN are as follows: the number of network layers is 5; the dimension of the hidden layer is 32; the learning rate is $1e-3$; the batch size is 128. For all variants of TreeNN $d_h = 256$. The different parameters of TreeNN-MP and TreeNN-con are as follows: the learning rate is $1e-3$; the batch size is 256. The different parameters of EQNET are as follows: the percentage of zero element in $\mathbf{n}$ is 0.61; the coefficient for SubexpAe, denoted by α, is $1-10^{-4t}$, where t stands for the epoch number; the learning rate is $1e-3$; the batch size is 256. The different parameters of TreeNN-inv are as follows: the coefficient of Equation (5), denoted by β, is 0.25; the learning rate is $1e-3$; the batch size is 128.

7.4 Comparison on Synthetic Dataset

Table 2 shows that all neural networks achieve much better performance than random guessing (50%), which indicates that neural networks indeed capture some inductive biases that are able to distinguish satisfiable formulae from unsatisfiable ones. However, different neural networks have different inductive biases, as reflected by their performance. Besides, Figure 3 shows that neural networks can generalize learned biases to larger formulae.

Transformer. The input of Transformer does not explicitly describe the recursive property of LTL, but is just a sequence of tokens. Although the multi-head self-attention mechanism is so powerful that it potentially learns the permutation invariance, and sequentiality, the learning cost of Transformer still affects its performance, making it perform the worst in three classes of neural networks. Note that Transformer runs fast thanks to the efficient parallel matrix operations of Transformer, and similar results are shown in Table 3.

RGCN. RGCN not only fails to keep the recursive property and permutation invariance but also cannot distinguish different operators. The poor performance of RGCN confirms that only modeling the permutation invariance is not sufficient for checking LTL satisfiability, which is different from the SAT problem.

TreeNN. Table 2 and Figure 3 show that all variants of TreeNN achieve the best performance. This may be due to the biggest difference between TreeNN and the other two classes of neural networks, *i.e.*, TreeNN keeps the recursive property of LTL. TreeNN-con is better than EQNET, which shows that the syntactic features help to check LTL satisfiability. TreeNN-con also outperforms TreeNN-MP. Although TreeNN-MP keeps the permutation invariance through the MP function, it compresses the information of sub-formulae, thus losing some important features. In contrast, TreeNN-con preserves complete information of sub-formulae via the concatenation function. TreeNN-inv performs slightly worse than TreeNN-con but is more efficient. These results, together with the comparison results between TreeNN-con and TreeNN-MP, demonstrate that the neural networks keeping the permutation invariance is less effective than keeping the sequentiality for checking LTL satisfiability.

Summary. Neural networks are capable to capture inductive biases for more accurately checking LTL satisfiability. Our experimental results demonstrate that the more a neural network keeps the logical properties of LTL, the more effective it is to capture the inductive biases that are beneficial to classification. These results answer the research questions **RQ 1** and **RQ 2**.

7.5 Comparison on Large Scale Dataset

For nuXmv and Aalta, we directly cite the results reported by the work [33]. The neural networks mainly use GPUs for computing, while the logical approaches use CPUs for computing. The ability of using GPUs is an advantage of neural networks. Overall, Table 4 shows that the four large scale datasets are challenging for logical approaches since there are still many formulae that nuXmv and Aalta cannot solve within the time limit. For every test set in our experiments, we directly remove formulae for which neither nuXmv nor Aalta can solve within the time limit, instead of reporting 3600 seconds, since we do not know the ground truth of them.

Performance. The performance of neural networks is surprisingly good but fluctuating. Overall, all variants of TreeNN achieve better accuracy and F1 score than Transformer and RGCN on all four large scale datasets. However, TreeNN-MP, TreeNN-con, and EQNET perform poorly in some datasets. It implies that the inductive biases captured by their different architectures vary in different datasets. But TreeNN-inv achieves good and stable performance by leveraging the inductive biases more properly. The above results imply that keeping the recursive property, permutation invariance, and sequentiality of LTL is effective for neural networks to generalize across different formula distributions.

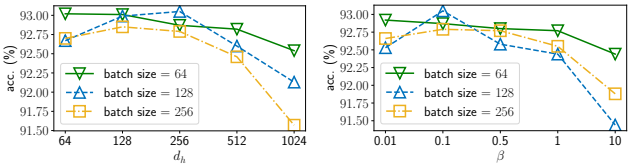
Running time. Overall, neural networks are much faster than logical approaches in most datasets. In particular, the running time of Aalta on *LTL-as-LTL$_f$*, *LTL$_f$-Specific*, and *DECLARE* is significantly longer than that of neural networks. This is not only due to the

Table 3: Evaluation results on the large scale datasets.

approach	*LTL-as-LTL$_f$*			*LTL$_f$-Specific*			*NASA-Boeing*			*DECLARE*		
	acc.	F1	time	acc.	F1	time	acc.	F1	time	acc.	F1	time
`nuXmv`	100.00	100.00	8,483.02	100.00	100.00	2,584.66	100.00	100.00	95.72	100.00	100.00	7,296.68
`Aalta`	100.00	100.00	352,224.95	100.00	100.00	1,148,839.93	100.00	100.00	9.37	100.00	100.00	356,043.51
`Transformer`	67.40	78.81	28.72	67.00	40.63	6.08	32.26	48.78	1.95	0.00	0.00	5.12
`RGCN`	73.39	83.31	6,021.31	91.00	88.69	3,145.00	37.10	54.12	220.10	0.00	0.00	5,250.53
`TreeNN-MP`	88.80	94.02	437.78	44.82	61.62	146.43	98.39	99.19	27.93	100.00	100.00	437.78
`TreeNN-con`	88.67	93.90	2,524.40	99.53	99.47	1,370.97	96.77	98.36	179.80	23.85	38.52	7,136.23
`EQNET`	86.37	92.58	2,481.76	98.76	98.62	1,404.96	53.23	69.47	193.44	96.33	98.13	7,182.01
`TreeNN-inv`	87.00	92.92	1,994.57	94.82	94.48	1,214.01	85.48	92.17	153.76	93.58	96.68	5,630.03

Table 4: The results of `nuXmv` and `Aalta` on the large scale datasets, where "size" means the number of formulae in the dataset, "# s." (*resp.* "# uns.") means the number of solved (*resp.* unsolved) formulae, and "both # uns." means the number of unsolved formulae that both `nuXmv` and `Aalta` failed to solve.

dataset	size	`nuXmv`		`Aalta`		both
		# s.	# uns.	# s.	# uns.	# uns.
LTL-as-LTL$_f$	7,442	7,422	20	7,326	116	19
LTL$_f$-Specific	1,700	1,700	0	1,381	319	0
NASA-Boeing	63	62	1	62	1	1
DECLARE	112	109	3	11	101	3



(a) Hyperparameters follow the optimal ones except d_h. (b) Hyperparameters follow the optimal ones except β.

Figure 4: Comparisons for different settings of d_h and β.

acceleration from GPUs, but also due to a theoretical guarantee that the computational cost of a neural network is linear with the size of the neural network.

Summary. The inductive biases captured by neural networks and the semantics of LTL may be different, as not all neural networks perform well on tests of generalization ability across different formula distributions. Nonetheless, the advantages exhibited by TreeNN still indicate that designing an architecture keeping logical properties, especially the recursive property, has a positive effect in improving the generalization ability. Besides, the results of `TreeNN-inv` confirm that neural networks have the potential to achieve a highly confident result for checking LTL satisfiability within a short time limit. These results answer the questions **RQ 2** and **RQ 3**.

8 THREAT TO VALIDITY

The primary threat to validity comes from different distributions of datasets. We have reason to believe that those approaches achieving generally higher accuracy on datasets with different distributions have better performance in practice. From the results in Table 2, Figure 3, and Table 3, `TreeNN-inv` has generally better accuracy, and it also has obvious speed advantages on large scale datasets. Therefore, we suggest that `TreeNN-inv` should be prioritized in practice. Although no neural network is guaranteed to work well on arbitrary datasets, we still show that it is promising to design a neural network that fulfill logical properties to improve generalization.

Hyperparameter selection is also a threat to validity. For all neural networks, we use grid search to find optimal hyperparameters. Due to limited space, we only analyze the hyperparameters of the best neural network `TreeNN-inv`. The main hyperparameters are d_h and β. We conduct experiments to show the impact of different setting of d_h and β on the test set of *SPOT*-$[100, 200)$, as illustrated in Figure 4. It can be seen that the accuracy remains rather stable for $d_h \in [64, 512]$ and $\beta \in [0.1, 0.5]$, which suggests that `TreeNN-inv` is rather robust on d_h but is probably sensitive to β.

9 CONCLUSION AND FUTURE WORK

LTL satisfiability checking is complex. Our main motivation is to explore a new paradigm for checking LTL satisfiability to outperform SOTA approaches. Our work shows that, by designing neural networks matching the recursive property, permutation invariance, and sequentiality, the neural networks can check LTL satisfiability, and can generalize across different sizes and different distributions of LTL formulae. Moreover, results on large scale datasets show that TreeNN is comparable with SOTA logical approaches in accuracy but is much more efficient. Our work makes it possible to obtain highly confident results for LTL satisfiability checking in polynomial time, which will benefit LTL-SAT-heavy tasks a lot.

Future work will extend our approach to validate the effectiveness of neural networks on intractable industrial instances and design more highly confident and verifiable end-to-end neural networks for checking LTL satisfiability.

ACKNOWLEDGMENTS

We thank Junhua Ma and Polong Chen for discussion on the paper and anonymous referees for helpful comments. This paper was supported by the National Natural Science Foundation of China (No. 61976232, 61876204, and 61906216), Guangdong Basic and Applied Basic Research Foundation (No. 2022A1515011355 and 2020A1515010642), Guangzhou Basic and Applied Basic Research Foundation (No. 2022A1515011355 and 2020A1515010642), Guangzhou Science and Technology Project (No. 202201011699), Guizhou Science Support Project (No. 2022-259), as well as Humanities and Social Science Research Project of Ministry of Education (No. 18YJCZH006).

REFERENCES

[1] Miltiadis Allamanis, Marc Brockschmidt, and Mahmoud Khademi. 2018. Learning to Represent Programs with Graphs. In *ICLR*.

[2] Miltiadis Allamanis, Pankajan Chanthirasegaran, Pushmeet Kohli, and Charles Sutton. 2017. Learning Continuous Semantic Representations of Symbolic Expressions. In *ICML*, Vol. 70. 80–88.

[3] Dalal Alrajeh, Jeff Kramer, Alessandra Russo, and Sebastin Uchitel. 2009. Learning operational requirements from goal models. In *ICSE*. 265–275.

[4] Saeed Amizadeh, Sergiy Matusevych, and Markus Weimer. 2019. Learning To Solve Circuit-SAT: An Unsupervised Differentiable Approach. In *ICLR*.

[5] Matteo Bertello, Nicola Gigante, Angelo Montanari, and Mark Reynolds. 2016. Leviathan: A New LTL Satisfiability Checking Tool Based on a One-Pass Tree-Shaped Tableau. In *IJCAI*. 950–956.

[6] Armin Biere, Alessandro Cimatti, Edmund M. Clarke, and Yunshan Zhu. 1999. Symbolic Model Checking without BDDs. In *TACAS*, Vol. 1579. 193–207.

[7] Marco Bozzano, Alessandro Cimatti, Anthony Fernandes Pires, David Jones, Greg Kimberly, T. Petri, R. Robinson, and Stefano Tonetta. 2015. Formal Design and Safety Analysis of AIR6110 Wheel Brake System. In *CAV*, Vol. 9206. 518–535.

[8] Aaron R. Bradley. 2011. SAT-Based Model Checking without Unrolling. In *VMCAI*, Vol. 6538. 70–87.

[9] Benedikt Bünz and Matthew Lamm. 2017. Graph Neural Networks and Boolean Satisfiability. *CoRR* abs/1702.03592 (2017). arXiv:1702.03592

[10] Jonathon Cai, Richard Shin, and Dawn Song. 2017. Making Neural Programming Architectures Generalize via Recursion. In *ICLR*.

[11] Chris Cameron, Rex Chen, Jason S. Hartford, and Kevin Leyton-Brown. 2020. Predicting Propositional Satisfiability via End-to-End Learning. In *AAAI*. 3324–3331.

[12] Roberto Cavada, Alessandro Cimatti, Michele Dorigatti, Alberto Griggio, Alessandro Mariotti, Andrea Micheli, Sergio Mover, Marco Roveri, and Stefano Tonetta. 2014. The nuXmv symbolic model checker. In *CAV*. 334–342.

[13] Claudio Di Ciccio, Fabrizio Maria Maggi, and Jan Mendling. 2016. Efficient discovery of Target-Branched Declare constraints. *Inf. Syst.* 56 (2016), 258–283.

[14] Alessandro Cimatti, Marco Pistore, Marco Roveri, and Roberto Sebastiani. 2002. Improving the Encoding of LTL Model Checking into SAT. In *VMCAI*, Vol. 2294. 196–207.

[15] Edmund M. Clarke, Armin Biere, Richard Raimi, and Yunshan Zhu. 2001. Bounded Model Checking Using Satisfiability Solving. *Formal Methods Syst. Des.* 19, 1 (2001), 7–34.

[16] Renzo Degiovanni, Facundo Molina, Germán Regis, and Nazareno Aguirre. 2018. A genetic algorithm for goal-conflict identification. In *ASE*. 520–531.

[17] Alexandre Duret-Lutz, Alexandre Lewkowicz, Amaury Fauchille, Thibaud Michaud, Etienne Renault, and Laurent Xu. 2016. Spot 2.0 - A Framework for LTL and ω -Automata Manipulation. In *ATVA*. 122–129.

[18] Niklas Eén, Alan Mishchenko, and Robert K. Brayton. 2011. Efficient implementation of property directed reachability. In *FMCAD*. 125–134.

[19] Richard Evans, David Saxton, David Amos, Pushmeet Kohli, and Edward Grefenstette. 2018. Can Neural Networks Understand Logical Entailment?. In *ICLR*.

[20] Valeria Fionda and Gianluigi Greco. 2018. LTL on Finite and Process Traces: Complexity Results and a Practical Reasoner. *J. Artif. Intell. Res.* 63 (2018), 557–623.

[21] Michael Fisher, Clare Dixon, and Martin Peim. 2001. Clausal temporal resolution. *ACM Trans. Comput. Log.* 2, 1 (2001), 12–56.

[22] Marco Gario, Alessandro Cimatti, Cristian Mattarei, Stefano Tonetta, and Kristin Yvonne Rozier. 2016. Model Checking at Scale: Automated Air Traffic Control Design Space Exploration. In *CAV*, Vol. 9780. 3–22.

[23] Giuseppe De Giacomo, Riccardo De Masellis, and Marco Montali. 2014. Reasoning on LTL on Finite Traces: Insensitivity to Infiniteness. In *AAAI*. 1027–1033.

[24] Giuseppe De Giacomo and Moshe Y. Vardi. 2013. Linear Temporal Logic and Linear Dynamic Logic on Finite Traces. In *IJCAI*. 854–860.

[25] Wenbo Guo, Dongliang Mu, Jun Xu, Purui Su, Gang Wang, and Xinyu Xing. 2018. LEMNA: Explaining Deep Learning based Security Applications. In *CCS*. 364–379.

[26] Christopher Hahn, Frederik Schmitt, Jens U. Kreber, Markus Norman Rabe, and Bernd Finkbeiner. 2021. Teaching Temporal Logics to Neural Networks. In *ICLR*.

[27] William L. Hamilton, Rex Ying, and Jure Leskovec. 2017. Representation Learning on Graphs: Methods and Applications. *IEEE Data Engineering Bulletin* 40, 3 (2017), 52–74.

[28] Jason S. Hartford, Devon R. Graham, Kevin Leyton-Brown, and Siamak Ravanbakhsh. 2018. Deep Models of Interactions Across Sets. In *ICML*, Vol. 80. 1914–1923.

[29] Yonit Kesten, Zohar Manna, Hugh McGuire, and Amir Pnueli. 1993. A Decision Algorithm for Full Propositional Temporal Logic. In *CAV*, Vol. 697. 97–109.

[30] Orna Kupferman and Moshe Y. Vardi. 1999. Model Checking of Safety Properties. In *CAV*, Vol. 1633. 172–183.

[31] Juho Lee, Yoonho Lee, Jungtaek Kim, Adam R. Kosiorek, Seungjin Choi, and Yee Whye Teh. 2019. Set Transformer: A Framework for Attention-based Permutation-Invariant Neural Networks. In *ICML*, Vol. 97. 3744–3753.

[32] Jianwen Li, Geguang Pu, Lijun Zhang, Moshe Y. Vardi, and Jifeng He. 2018. Accelerating LTL satisfiability checking by SAT solvers. *J. Log. Comput.* 28, 6 (2018), 1011–1030.

[33] Jianwen Li, Geguang Pu, Yueling Zhang, Moshe Y. Vardi, and Kristin Y. Rozier. 2020. SAT-based explicit LTLf satisfiability checking. *Artif. Intell.* 289 (2020), 103369.

[34] Jianwen Li, Yinbo Yao, Geguang Pu, Lijun Zhang, and Jifeng He. 2014. Aalta: an LTL satisfiability checker over Infinite/Finite traces. In *FSE*. 731–734.

[35] Jianwen Li, Lijun Zhang, Geguang Pu, Moshe Y. Vardi, and Jifeng He. 2013. LTL Satisfiability Checking Revisited. In *TIME*. 91–98.

[36] Jianwen Li, Lijun Zhang, Shufang Zhu, Geguang Pu, Moshe Y. Vardi, and Jifeng He. 2018. An explicit transition system construction approach to LTL satisfiability checking. *Formal Aspects Comput.* 30, 2 (2018), 193–217.

[37] Jianwen Li, Shufang Zhu, Geguang Pu, and Moshe Y. Vardi. 2015. SAT-Based Explicit LTL Reasoning. In *HVC*, Vol. 9434. 209–224.

[38] Jianwen Li, Shufang Zhu, Geguang Pu, Lijun Zhang, and Moshe Y. Vardi. 2019. SAT-based explicit LTL reasoning and its application to satisfiability checking. *Formal Methods in System Design* 54, 2 (2019), 164–190.

[39] Weilin Luo, Hai Wan, Xiaotong Song, Binhao Yang, Hongzhen Zhong, and Yin Chen. 2021. How to Identify Boundary Conditions with Contrasty Metric?. In *ICSE*. 1473–1484.

[40] Weilin Luo, Hai Wan, Hongzhen Zhong, Ou Wei, Biqing Fang, and Xiaotong Song. 2021. An Efficient Two-phase Method for Prime Compilation of Non-clausal Boolean Formulae. In *ICCAD*. 1–9.

[41] Mingjun Ma, Dehui Du, Yuanhao Liu, Yanyun Wang, and Yiyang Li. 2022. Efficient Adversarial Sequence Generation for RNN with Symbolic Weighted Finite Automata. In *SafeAI@AAAI*, Vol. 3087.

[42] Fabrizio Maria Maggi, Marlon Dumas, Luciano García-Bañuelos, and Marco Montali. 2013. Discovering Data-Aware Declarative Process Models from Event Logs. In *BPM*, Vol. 8094. 81–96.

[43] Nicolas Markey and Philippe Schnoebelen. 2003. Model Checking a Path. In *CONCUR*, Vol. 2761. 248–262.

[44] Kenneth L. McMillan. 2003. Interpolation and SAT-Based Model Checking. In *CAV*, Vol. 2725. 1–13.

[45] Ryan L. Murphy, Balasubramaniam Srinivasan, Vinayak A. Rao, and Bruno Ribeiro. 2019. Janossy Pooling: Learning Deep Permutation-Invariant Functions for Variable-Size Inputs. In *ICLR*.

[46] Aditya Paliwal, Sarah M. Loos, Markus N. Rabe, Kshitij Bansal, and Christian Szegedy. 2020. Graph Representations for Higher-Order Logic and Theorem Proving. In *AAAI*. 2967–2974.

[47] Amir Pnueli. 1977. The Temporal Logic of Programs. In *FOCS*. 46–57.

[48] Johannes Prescher, Claudio Di Ciccio, and Jan Mendling. 2014. From Declarative Processes to Imperative Models. In *SIMPDA*, Vol. 1293. 162–173.

[49] Marco Túlio Ribeiro, Sameer Singh, and Carlos Guestrin. 2016. "Why Should I Trust You?": Explaining the Predictions of Any Classifier. In *KDD*. 1135–1144.

[50] Kristin Y. Rozier and Moshe Y. Vardi. 2007. LTL Satisfiability Checking. In *SPIN*, Vol. 4595. 149–167.

[51] Kristin Y. Rozier and Moshe Y. Vardi. 2010. LTL satisfiability checking. *International Journal on Software Tools for Technology Transfer* 12, 2 (2010), 123–137.

[52] Michael Sejr Schlichtkrull, Thomas N. Kipf, Peter Bloem, Rianne van den Berg, Ivan Titov, and Max Welling. 2018. Modeling Relational Data with Graph Convolutional Networks. In *ESWC*, Vol. 10843. 593–607.

[53] Viktor Schuppan. 2009. Towards a Notion of Unsatisfiable Cores for LTL. In *FSEN*, Vol. 5961. Springer, 129–145.

[54] Viktor Schuppan. 2016. Enhancing unsatisfiable cores for LTL with information on temporal relevance. *Theor. Comput. Sci.* 655 (2016), 155–192.

[55] Viktor Schuppan. 2016. Extracting unsatisfiable cores for LTL via temporal resolution. *Acta Informatica* 53, 3 (2016), 247–299.

[56] Viktor Schuppan and Luthfi Darmawan. 2011. Evaluating LTL Satisfiability Solvers. In *ATVA*, Vol. 6996. 397–413.

[57] Stefan Schwendimann. 1998. A New One-Pass Tableau Calculus for PLTL. In *TABLEAUX*, Vol. 1397. 277–292.

[58] Daniel Selsam and Nikolaj Bjørner. 2019. Guiding High-Performance SAT Solvers with Unsat-Core Predictions. In *SAT*, Vol. 11628. 336–353.

[59] Daniel Selsam, Matthew Lamm, Benedikt Bünz, Percy Liang, Leonardo de Moura, and David L. Dill. 2019. Learning a SAT Solver from Single-Bit Supervision. In *ICLR*. 1–11.

[60] Xujie Si, Hanjun Dai, Mukund Raghothaman, Mayur Naik, and Le Song. 2018. Learning Loop Invariants for Program Verification. In *NeurIPS*. 7762–7773.

[61] A. Prasad Sistla and Edmund M. Clarke. 1985. The Complexity of Propositional Linear Temporal Logics. *J. ACM* 32, 3 (1985), 733–749.

[62] Richard Socher, Brody Huval, Christopher D. Manning, and Andrew Y. Ng. 2012. Semantic Compositionality through Recursive Matrix-Vector Spaces. In *EMNLP-CoNLL*. 1201–1211.

[63] Richard Socher, Alex Perelygin, Jean Wu, Jason Chuang, Christopher D. Manning, Andrew Y. Ng, and Christopher Potts. 2013. Recursive Deep Models for Semantic Compositionality Over a Sentiment Treebank. In *EMNLP*. 1631–1642.

[64] Pashootan Vaezipoor, Andrew C. Li, Rodrigo Toro Icarte, and Sheila A. McIlraith. 2021. LTL2Action: Generalizing LTL Instructions for Multi-Task RL. In *ICML*, Vol. 139. 10497–10508.
[65] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. 2017. Attention is All you Need. In *NeurIPS*. 5998–6008.
[66] Pierre Wolper. 1985. The tableau method for temporal logic: An overview. *Logique et Analyse* (1985), 119–136.
[67] Martin De Wulf, Laurent Doyen, Nicolas Maquet, and Jean-François Raskin. 2008. Antichains: Alternative Algorithms for LTL Satisfiability and Model-Checking. In *TACAS*, Vol. 4963. 63–77.
[68] Yaqi Xie, Fan Zhou, and Harold Soh. 2021. Embedding Symbolic Temporal Knowledge into Deep Sequential Models. In *ICRA*. 4267–4273.
[69] Keyulu Xu, Jingling Li, Mozhi Zhang, Simon S. Du, Ken-ichi Kawarabayashi, and Stefanie Jegelka. 2020. What Can Neural Networks Reason About?. In *ICLR*.
[70] Manzil Zaheer, Satwik Kottur, Siamak Ravanbakhsh, Barnabás Póczos, Ruslan Salakhutdinov, and Alexander J. Smola. 2017. Deep Sets. In *NeurIPS*. 3391–3401.
[71] Wenjie Zhang, Zeyu Sun, Qihao Zhu, Ge Li, Shaowei Cai, Yingfei Xiong, and Lu Zhang. 2020. NLocalSAT: Boosting Local Search with Solution Prediction. In *IJCAI*. 1177–1183.
[72] Xiyue Zhang, Xiaoning Du, Xiaofei Xie, Lei Ma, Yang Liu, and Meng Sun. 2021. Decision-Guided Weighted Automata Extraction from Recurrent Neural Networks. In *AAAI*. 11699–11707.
[73] Yan Zhang, Jonathon S. Hare, and Adam Prügel-Bennett. 2020. FSPool: Learning Set Representations with Featurewise Sort Pooling. In *ICLR*.