Transformers Generalize to the Semantics of Logics

Christopher Hahn
Saarland University
66123 Saarbrücken, Germany
hahn@react.uni-saarland.de

Frederik Schmitt
Saarland University
66123 Saarbrücken, Germany
s8fkschm@stud.uni-saarland.de

Jens U. Kreber
Saarland University
66123 Saarbrücken, Germany
kreber@react.uni-saarland.de

Markus N. Rabe
Google Research
Mountain View, CA, USA
mrabe@google.com

Bernd Finkbeiner
CISPA Helmholtz Center for Information Security
66123 Saarbrücken, Germany
finkbeiner@cispa.saarland

Abstract

We show that neural networks can learn the semantics of propositional and linear-time temporal logic (LTL) from imperfect training data. Instead of only predicting the truth value of a formula, we use a Transformer architecture to predict the solution for a given formula, e.g., a variable assignment for a formula in propositional logic. Most formulas have many solutions and the training data thus depends on the particularities of the generator. We make the surprising observation that while the Transformer does not perfectly predict the generator's output, it still produces correct solutions to almost all formulas, even when its prediction deviates from the generator. It appears that it is easier to learn the semantics of the logics than the particularities of the generator. We observe that the Transformer preserves this semantic generalization even when challenged with formulas of a size it has never encountered before. Surprisingly, the Transformer solves almost all LTL formulas in our test set including those for which our generator timed out.

1 Introduction

Machine learning has revolutionized several areas of computer science, achieving human-level performance in tasks like image recognition [19], face recognition [44], translation [50, 45], and board games [28, 40]. For complex tasks that involve symbolic reasoning, however, deep learning techniques are often considered as insufficient. Applications of machine learning in logical reasoning problems are therefore few, and mostly restricted to sub-problems within larger logical frameworks, such as computing heuristics in solvers [24, 3, 35] or predicting individual proof steps [4, 14].

In this paper, we study if neural networks can solve difficult logical problems directly, without any external reasoning, and we analyze how well neural networks generalize from imperfect training data to the underlying semantics of the logics. As a problem that requires deep understanding of the logical semantics, we apply deep learning to the problem of computing a solution to a logical formula. For example, given a formula of propositional logic, we are interested in finding a satisfying assignment to the propositional variables. Earlier work tackled the satisfiability problem of propositional logic by framing it as a classification problem [36]. Their neural network architecture is explicitly crafted

Preprint. Under review.

for propositional formulas given in conjunctive normal form (CNF). By contrast, we use a generic sequence-to-sequence model, namely the Transformer, to translate logical formulas into satisfying assignments. The key advantage of our approach is that there is no need to handcraft the architecture: it works as-is, even for logics with substantially different semantical concepts. We demonstrate this flexibility with two very popular logics: general propositional logic (without restriction to CNF) and linear-time temporal logic (LTL). For both logics, the Transformer indeed generalizes to the semantics of the logics and solves intricate reasoning problems.

Satisfiability solving for propositional logic has numerous applications throughout computer science. In computer-aided verification, to name just one prominent example, propositional SAT solvers are the core reasoning engines for problems like SMT [8, 5], fault diagnosis [42], bounded model checking [7], and bounded synthesis [13].

Linear-time temporal logic (LTL) [31] is the most common logic for the specification of reactive systems, and the basis for industrial hardware specification languages like the IEEE standard PSL [22]. LTL has temporal operators that reason about infinite sequences. In verification, this is frequently used to specify temporal aspects of executions of software and hardware systems. For example, consider the following specification of an arbiter: $\square(\texttt{request} \rightarrow \diamondsuit\texttt{grant})$ states that, at every point in time ($\square$-operator), if there is a request signal, then a grant signal must follow at some future point in time ($\diamondsuit$-operator). Understanding LTL is an even more significant challenge for a deep neural network than propositional logic, because an LTL formula asks not only for a single assignment of boolean values to a set of propositional variables, but rather for a possibly infinite sequence of such assignments. This is also reflected in the complexity of the satisfiability problems: satisfiability of propositional logic is NP-complete, satisfiability of LTL is PSPACE-complete.

For the generation of the training data, we use standard DPLL and automata-based algorithms. Our main contribution is the surprising finding that while the Transformer does not perfectly predict the generator's output, it still produces correct solutions to almost all formulas, even when its prediction deviates from the generator. This suggests that the models must have learned the semantics of logics instead of trying to match the particular choices of the generators. For example, for propositional logic, our best performing model predicts the exact output of the generator in 58.1% of the cases, and produces correct assignments in 96.5% of the cases on a held-out test set. This semantic understanding is preserved even when the Transformer is challenged with formulas of a size it has never seen before. We furthermore make the following two contributions. We show that with a positional encoding for *trees* [39], the Transformer generalizes to longer formulas than seen during training: we trained a Transformer solely on LTL formulas of length up to 35. With the standard positional encoding, it achieves a total accuracy of only 40.5% on formulas of size 36 to 50. With the positional encoding for trees, it reaches 92.2% accuracy (for details see Section 5.2). Additionally, the Transformer can solve almost all hard LTL formulas in our test set for which even our generator timed out.

The paper is structured as follows. We describe the problem definitions in Section 2. We present our data generation in Section 3. Our experimental setup is described in Section 4 and our findings in Section 5. We give an overview over related work in Section 6 before concluding in Section 7.

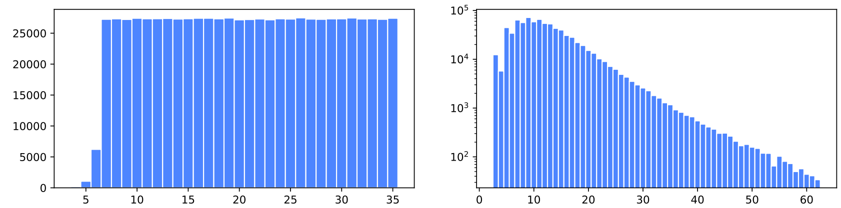
2 Problem Definition

We apply deep learning to the problem of computing a solution to a logical formula. For propositional logic, a solution is an assignment to the propositional variables that satisfies the formula; for LTL, a solution is a sequence of assignments, called *trace*, that satisfies the formula. In general, such solutions are not necessarily unique. To address this issue, we keep the solutions to the formulas in our data sets as general as possible, i.e., we allow for *partial* assignments and *symbolic* traces, which are defined in the following.

2.1 Assignment Generation for Propositional Logic

A propositional formula consists of Boolean operators $\wedge$ (and), $\vee$ (or), $\neg$ (not), and variables also called literals or propositions. We consider the derived operators $\varphi_1 \rightarrow \varphi_2 \equiv \neg\varphi_1 \vee \varphi_2$ (implication), $\varphi_1 \leftrightarrow \varphi_2 \equiv (\varphi_1 \rightarrow \varphi_2) \wedge (\varphi_2 \rightarrow \varphi_1)$ (equivalence), and $\varphi_1 \oplus \varphi_2 \equiv \neg(\varphi_1 \leftrightarrow \varphi_2)$ (xor).

Given a propositional Boolean formula φ, the satisfiability problem asks if there exists a Boolean assignment $\Pi : \mathcal{V} \mapsto \mathbb{B}$ for every literal in φ such that φ evaluates to $true$. For example, consider the



(a) Formula distribution by size. (b) Trace distribution by size.

Figure 1: Size distributions in the LTL_{35} training set: on the x-axis is the size of the formulas/traces; on the y-axis the number of formulas/traces.

following propositional formula, given in conjunctive normal form (CNF): $(x_1 \vee x_2 \vee \neg x_3) \wedge (\neg x_1 \vee x_3)$. A possible satisfying assignment for this formula would be $\{(x_1, \mathit{true}), (x_2, \mathit{false}), (x_3, \mathit{true})\}$. To be as general as possible, we allow a satisfying assignment to be *partial*, i.e., if the truth value of a propositions can be arbitrary, it will be omitted. For example, $\{(x_1, \mathit{true}), (x_3, \mathit{true})\}$ would be a satisfying partial assignment for the formula above. We define a *minimal unsatisfiable core* of an unsatisfiable formula φ, given in CNF, as an unsatisfiable subset of clauses φ_{core} of φ, such that every proper subset of clauses of φ_{core} is still satisfiable.

2.2 Trace Generation for Linear-time Temporal Logic

Linear-time temporal logic (LTL) [31] combines the propositional connectives, described above, with temporal operators such as the *Next* operator $\bigcirc$ and the *Until* operator $\mathcal{U}$. $\bigcirc \varphi$ means that φ holds in the *next* position of a sequence; $\varphi_1 \,\mathcal{U}\, \varphi_2$ means that φ_1 holds *until* φ_2 holds. There are several derived operators, such as $\Diamond \varphi \equiv \mathit{true} \,\mathcal{U}\, \varphi$ and $\Box \varphi \equiv \neg \Diamond \neg \varphi$. $\Diamond \varphi$ states that φ will *eventually* hold in the future and $\Box \varphi$ states that φ holds *globally*. Operators can be nested: $\Box \Diamond \varphi$, for example, states that φ has to occur infinitely often. The full semantics of LTL can be found in Appendix A.

We consider infinite sequences over sets of atomic propositions. We call such sequences *explicit traces*. In each position, an explicit trace defines an assignment to the propositions, where all propositions that occur in the set evaluate to true, all others to false. We define a *symbolic trace* as a sequence of propositional formulas over the atomic propositions. A symbolic trace t_s defines the set $\mathit{Sequences}(t_s)$ of explicit sequences t where $t[i]$ satisfies $t_s[i]$ for all i. Symbolic traces allow us to underspecify propositions when they do not matter. For example, the LTL formula $\bigcirc \Box a$ over atomic propositions a and b is satisfied by the symbolic trace: $\mathit{true}(a)^\omega$, which leaves open whether a holds on the first position as well.

We say an explicit trace t is an *instance* of a symbolic trace t_s if $t \in \mathit{Sequences}(t_s)$. For example, given $AP = \{a, b, c\}$, the symbolic trace $(a \wedge b)^\omega$ defines the infinite set of explicit traces $\{\alpha^\omega \mid \forall i \in \mathbb{N}.\ a, b \in \alpha[i]\}$. Traces $t_c = \{a, b, c\}^\omega$ and $t_{\neg c} = \{a, b\}^\omega$ are two of the infinitely many instances of the symbolic trace $(a \wedge b)^\omega$, i.e., $t_c, t_{\neg c} \in (a \wedge b)^\omega$. Given a satisfiable LTL formula φ_{sat}, the symbolic trace generation problem of LTL asks for a symbolic trace t_s such that every instance of t_s satisfies the formula, i.e., $\forall t \in \mathit{Sequences}(t_s) :\ t \models \varphi_{sat}$. Traces described by LTL formulas are infinitely long. For satifiability, it suffices however to consider ultimately periodic traces, i.e., traces that are finitely represented in the form of a "lasso" uv^ω, where u and v are finite sequences of sets of atomic propositions.

3 Data Sets

Our data sets contain 1 million formulas, each together with a solution, i.e., a satisfying partial assignment for propositional logic and a satisfying symbolic trace for LTL. Unless stated otherwise, the number of different propositions is fixed to 5 for both propositional logic and LTL. The data sets differ in the maximum size of the formula's syntax tree. We refer to the data sets as follows: Prop_{35}, for example, corresponds to 1M propositional formulas of maximum size 35 and their partial assignments and LTL_{35} corresponds to 1M LTL formulas of maximum size 35 and their satisfying symbolic traces. Each data set is split into a training set of 800K formulas, a validation set of 100K formulas, and a test set of 100K formulas. All data sets are uniformly distributed in size, apart from the lower-sized end due to the limited number of unique small formulas. Figure 1 exemplarily shows the formula and trace distribution of the data set LTL_{35}.

To generate the formulas, we used the `randltl` tool of the `spot` framework [10], which builds unique formulas in a specified size interval, following a supplied node probability distribution. Note that during the building process, the actual distribution occasionally differs from the given distribution in order to meet the size constraints, e.g., by masking out all binary operators. The distribution between all k-ary nodes always remains the same. To furthermore achieve a (quasi) uniform distribution in size, we subsequently filtered the generated formulas.

We tested our best performing models on an infix and a Polish notation of the formulas and found that it had no significant impact on the performance of the Transformer. We decided to use the Polish notation, because it allowed us to drop parentheses. In the following, we describe the data sets.

3.1 Propositional Logic: Assignment Generation

For the generation of propositional formulas, the specified node distribution puts equal weight on $\wedge$, $\vee$, and $\neg$ operators and half as much weight on the derived operators $\leftrightarrow$ and $\oplus$ individually. In contrast to previous work [36], which is restricted to formulas in CNF, we allow an arbitrary formula structure and derived operators.

A satisfying assignment is represented as an alternating sequence of propositions and truth values, given as 0 and 1. The sequence $a0b1c0$, for example, represents the partial assignment $\{(a, \mathit{false}), (b, \mathit{true}), (c, \mathit{false})\}$, meaning that the truth values of propositions d and e can be chosen arbitrarily (note that we allow five propositions). We used pyaiger [46], which builds on Glucose 4 [2] as its underlying SAT solver. We construct the partial assignments with a standard method in SAT solving: We query the SAT solver for a minimal unsatisfiable core of the *negation* of the formula. To give the interested reader an idea of the level of difficulty of the data set, the following table shows three random examples from our training set $Prop_{35}$. The first line shows the formula and the assignment in mathematical notation. The second line shows the syntactic representation:

<table>
<tr><th>propositional formula</th><th>satisfying partial assignment</th></tr>
<tr><td>$((d \wedge \neg e) \wedge (\neg a \vee \neg e)) \leftrightarrow ((\neg \oplus (\neg b \leftrightarrow \neg e))$
$\vee((e \oplus (b \wedge d)) \oplus \neg(\neg c \vee (\neg a \leftrightarrow e))))$
<->&&d!e|!a!e|xor!b<->!b!exorxore&bd!|!c<->!ae</td><td>$\{(a,0),(b,0),(c,1),(d,1),(e,0)\}$

a0b0c1d1e0</td></tr>
<tr><td>$(c \vee e) \vee (\neg a \leftrightarrow \neg b)$
||ce<->!a!b</td><td>$\{(c,1)\}$
c1</td></tr>
<tr><td>$\neg((b \vee e) \oplus ((\neg a \vee (\neg d \leftrightarrow \neg e))$
$\vee(\neg b \vee (((\neg a \wedge b) \wedge \neg b) \wedge d))))$
!xor!be||!a<->!d!e!|!b&&&!ab!b!d</td><td>$\{(d,1),(e,1)\}$

d1e1</td></tr>
</table>

3.2 LTL: Trace Generation

For the generation of LTL formulas, our node distribution puts equal weight on all operators $\neg, \wedge, \bigcirc$ and $\mathcal{U}$. Constants `True` and `False` are allowed with 2.5 times less probability than propositions. We use a compact syntax for ultimately periodic symbolic traces: Each position in the trace is separated by the delimiter ";". True and False are represented by "1" and "0", respectively. The beginning of the period v is signaled by the character "{" and analogously its end by "}". For example, the ultimately periodic symbolic trace denoted by $a; a; a; \{b\}$, describes all infinite traces where on the first 3 positions a must hold followed by an infinite period on which b must hold on every position.

Given a satisfiable LTL formula φ, our trace generator constructs a Büchi automaton A_φ that accepts exactly the language defined by the LTL formula, i.e., $\mathcal{L}(A_\varphi) = \mathcal{L}(\varphi)$. From this automaton, we construct an arbitrary accepted symbolic trace, by searching for an accepting run in A_φ. We use `spot` [10] for the manipulation of LTL formulas and automata over infinite sequences.

See Figure 1 for the size distribution of generated examples in LTL_{35}. Note that we filtered out examples with traces larger than 62 (less than 0.05% of the original set). In the following table, we illustrate our data set with three random examples from training set LTL_{35}. The first line shows the LTL formula and the symbolic trace in mathematical notation. The second line shows the syntactic representation:

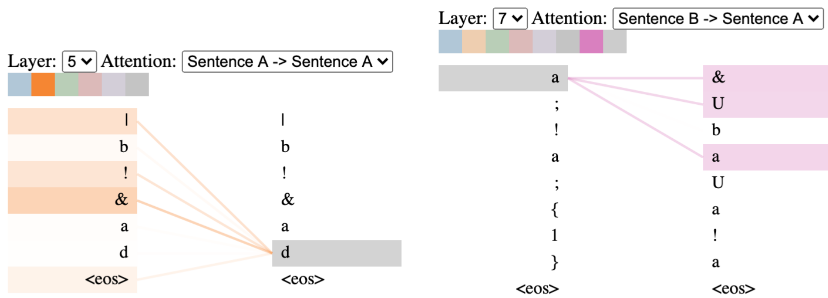


Figure 2: Self-attention of the example propositional formula $b \vee \neg(a \wedge d)$ in data set $Prop_{35}$ (left). Encoder-decoder-attention of the example LTL formula $(b\,\mathcal{U}\,a) \wedge (a\,\mathcal{U}\,\neg a)$ in data set LTL_{35} (right).

LTL formula	satisfying symbolic trace
$\bigcirc((d\,\mathcal{U}\,c)\,\mathcal{U}\bigcirc\bigcirc d)\wedge\bigcirc(b\wedge\neg(\neg d\,\mathcal{U}\,c))$ `&XUUdcXXdX&b!U!dc`	$\mathit{true}\ (b\wedge\neg c\wedge\neg d)\ (\neg c\wedge d)\ d\ (\mathit{true})^\omega$ `1;&&b!c!d;&!cd;d;{1}`
$\neg\bigcirc((\bigcirc e\wedge(\mathit{true}\,\mathcal{U}\,b)\wedge\bigcirc c)\,\mathcal{U}\,c)$ `!XU&&XeU1bXcc`	$\mathit{true}\ (\neg b\wedge\neg c)\ (\neg b)^\omega$ `1;&!b!c;{!b}`
$\bigcirc\neg((\neg c\wedge d)\,\mathcal{U}\bigcirc d)$ `X!U&!cdXd`	$\mathit{true}\ (c\vee\neg d)\ (\neg d)\ (\mathit{true})^\omega$ `1;|c!d;!d;{1}`

4 Experimental Setup

We have implemented the Transformer architecture [45].[1] Our implementation processes the input and output sequences token-by-token. We trained all Transformers on a single NVIDIA P100 GPU once with different hyperparameters. All training has been done with a dropout rate of 0.1 and early stopping on the validation set. Note that the embedding size will automatically be floored to be divisible by the number of attention heads. The training of the models took between 2 and 16 hours. For the output decoding, we used a *beam* search [49], a heuristic best-first search that solely keeps track of a predetermined number of best partial solutions. We used a beam size between 2 and 3 and an α of 1.

Evaluation method. Since the solution of a logical formula is not necessarily unique, we use two different measures of accuracy to evaluate the generalization to the semantics of the logics: we distinguish between the accuracy of an *exact syntactic match* and the *semantic* accuracy. We refer to the fraction of formulas that is translated into an assignment or a trace that is syntactically the same as the target in our data set as the accuracy of *exact syntactic matches*. Assignment and traces that are not syntactically equivalent to the ones chosen in the data can still satisfy the given formula. We define the *total accuracy* as the fraction of assignment and traces that satisfy the given formula. We also simply refer to this as a correct assignment or a correct trace. We denote the subtraction of the overall accuracy and the accuracy of an exact syntactic match as the *semantic* accuracy.

5 Experimental Results and Discussion

In this section, we describe our experimental results and discuss the generalization to the logical semantics in detail. We consider propositional logic first. Our experiments show that the Transformer learns the underlying semantics and also generalizes to larger formulas than it has seen during training. Secondly, we find that the Transformer also generalizes to the semantics of the more expressive logics, namely linear-time temporal logic (LTL). Despite the complex temporal operators, the Transformer also generalizes to larger formulas than seen during training when utilizing a tree positional encoding. Lastly, and most surprisingly, the Transformer even solves the trace generation problem in 199 out of 201 cases for formulas on which our generator timed out.

[1] The code, our data sets, and data generators are available at `https://github.com/reactive-systems/deepltl`.

Table 1: Exact syntactic match and total accuracy of different Transformers, tested on LTL_{35}: Layers refer to the size of the encoder and decoder stacks; Heads refer to the number of attention heads; FC size refers to the size of the fully-connected neural networks inside the encoder and decoders.

Embedding size	Layers	Heads	FC size	Batch Size	Train Steps	Exact match	Correct
128	3	4	512	512	45K	78.0%	97.1%
128	5	2	512	512	45K	80.4%	97.4%
128	5	4	256	512	45K	81.0%	97.4%
128	5	4	512	512	45K	82.0%	97.9%
128	5	4	1024	512	45K	80.3%	97.3%
128	5	6	1024	512	45K	81.8%	97.7%
128	5	8	512	512	45K	82.0%	97.8%
128	5	8	1024	512	45K	82.5%	97.9%
128	5	8	1500	512	45K	82.6%	97.8%
128	5	12	1024	512	45K	81.9%	97.5%
128	8	4	512	512	45K	83.2%	98.3%
128	8	8	1024	768	50K	83.8%	**98.5%**
128	10	4	512	512	75K	82.9%	97.6%
256	5	4	512	512	45K	82.3%	97.9%

5.1 Propositional Logic

As a baseline for our generalization experiments, we trained the following Transformer on $Prop_{35}$:

Embedding size	Layers	Heads	FC size	Batch Size	Train Steps	Exact match	Correct
enc:128, dec:64	6	6	512	1024	50K	58.1%	**96.5%**

We observe a striking 38.4% gap between predictions that were exact syntactic matches of our DPLL-based generator and correct predictions of the Transformer. Only 3.5% of the time, the Transformer outputs an incorrect assignment. Note that we allow the derived operators $\oplus$ and $\leftrightarrow$ in these experiments, which succinctly represent complicated logical constructs.

For example, the formula $b \vee \neg(a \wedge d)$ occurs in our data set $Prop_{35}$ and its corresponding assignment is $\{(a, 0)\}$. The Transformer, however, outputs `d0`, i.e., it goes with the assignment of setting d to $false$, which is also a correct solution. Figure 2 displays a visualization [47] of one encoder self-attention head on this example. While encoding d, the model pays the most attention to the closest operator ($\wedge$), but also to the top-level operators ($\vee$ and $\neg$). When the formulas get larger, the solutions where the Transformer differs from the DPLL algorithm accumulate. Consider, for example, the formula $\neg b \vee (e \leftrightarrow b \vee c \vee \neg d) \vee (c \wedge (b \oplus (a \oplus \neg d)) \oplus (\neg c \leftrightarrow d) \wedge (a \leftrightarrow (b \oplus (b \oplus e))))$, which is also in the data set $Prop_{35}$. The generator suggests the assignment $\{(a, 1), (c, 1), (d, 0)\}$. The Transformer, however, outputs $e0$, i.e., the singleton assignment of setting e to $false$, which turns out to be a (very small) solution as well.

We only achieved stable training in this experiment by setting the decoder embedding size significantly lower to either 64 or even 32. Keeping the decoder embedding size at 128 led to very unstable training.

Semantic generalization to larger formulas. In our next experiment, we tested whether this generalization to the semantic is preserved when the Transformer encounters formulas of a larger size than it ever saw during training. The generalization to larger formulas significantly increases by utilizing a positional encoding based on the tree representation of the formula [39]. By encoding the position in the tree representation, the positional encoding represents the tree structure rather than just the order of the input sequence. This allows for learning tree-based relationships such as child, parent, or cousin node. When challenged with formulas of size 35 to 50, our best performing Transformer (see above) trained on $Prop_{35}$ achieves an exact syntactic match of 35.8% and an overall accuracy of 86.1%. In comparison, without the tree positional encoding, the Transformer achieves a syntactic match of only 29.0% and an overall accuracy of only 75.7%. Note that both positional encodings work equally well when not considering larger formulas.

Training on larger formulas without derived operators. The focus of our experiments lies on the generalization to the semantics of propositional logic with arbitrary operators. For the sake of completeness, we also ran experiments on training data without derived operators. We generated $1M$

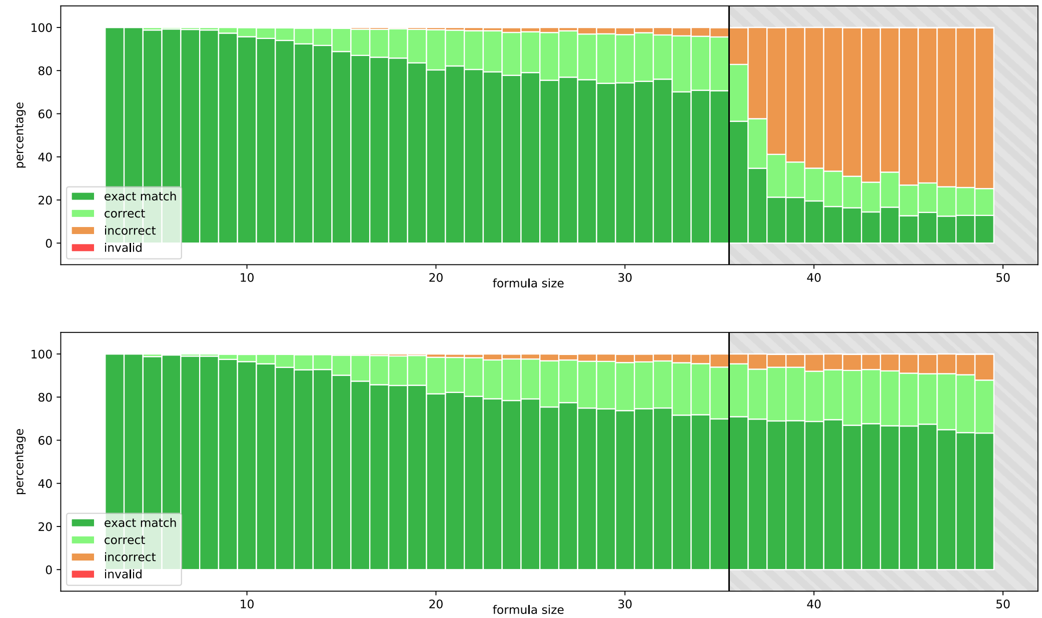


Figure 3: Performance of our best model (only trained on LTL_{35}) on LTL_{50} with a standard positional encoding (top) and a tree positional encoding (bottom). Exact syntactic matches are displayed in green, the semantic accuracy in light green and the incorrect predictions in yellow. The shaded area indicates the formula sizes the model was not trained on.

propositional formulas with 10 (instead of 5) different propositions and a formula size of maximal 60 (instead of 35), without $\oplus$ and $\leftrightarrow$. The Transformer performs even better on such large formulas:

Embedding size	Layers	Heads	FC size	Batch Size	Train Steps	Exact match	Correct
enc:128, dec:64	5	4	512	1300	21.5K	57.4%	**99.2%**

We conclude that, also for the Transformer, the succinctness of logical formulas, i.e., allowing succinct operators like $\oplus$ and $\leftrightarrow$, contributes heavily to the difficulty of the problem.

5.2 Linear-time Temporal Logic

In the following, we report on experiments that show that the Transformer can also generalize to the semantics of more complex and more expressive logics than propositional logic. We consider an example first. The LTL formula $(b\,\mathcal{U}\,a) \wedge (a\,\mathcal{U}\,\neg a)$ states that b has to hold along the trace until a holds and a has to hold until a does not hold anymore. The automaton-based generator suggests the trace $(\neg a \wedge b)\ a\ (true)^{\omega}$, i.e., to first satisfy the second until by immediately disallowing a. The satisfaction of the first until is then postponed to the second position of trace, which forces b to hold on the first position. The Transformer, however, chooses the following more general trace $a\ (\neg a)\ (true)^{\omega}$, by satisfying the until operators in order (see Figure 2).

The *unshaded* part of Figure 3 displays the performance of our best model on the LTL_{35} data set for both positional encodings. Note that the Transformers were solely trained on formulas of size less or equal to 35. We observe that in this range the exact syntactic accuracy decreases when the formulas grow in size. The overall accuracy, however, stays high. With the standard positional encoding, for example, the model achieves an exact syntactic accuracy of 83.8% and a total accuracy of 98.5% on LTL_{35}, i.e., in 14.7% of the cases, the Transformer deviates from our automaton-based data generator. The evolution of the exact syntactic matches and the semantic accuracy during training can be found in Appendix B. An analysis on typical handcrafted formula examples can be found in Appendix C.

Hyperparameter analysis. Table 1 shows the effect of the most significant parameters on the performance of Transformers. The performance largely benefits from an increased number of layers, with 8 yielding the best results. Increasing the number further, even with much more training time, did not result in better or even led to worse results. A slightly less important role plays the number of heads and the dimension of the intermediate fully-connected feed-forward networks (FC). While a certain FC size is important, increasing it alone will not improve results. Changing the number of heads alone has also almost no impact on performance. Increasing both simultaneously, however, will result in a small gain. This seems reasonable, since more heads can provide more distinct information to the subsequent processing by the fully-connected feed-forward network. Increasing the embeddings size from 128 to 256 very slightly improves the exact match accuracy. But likewise it also degrades the total accuracy, so we therefore stuck with the former setting.

Semantic generalization to larger formulas. We tested how well the Transformer generalizes to LTL formulas of a size it has never seen before. We trained on LTL_{35} and observed the performance on LTL_{50}. The the exact syntactic matches and total accuracy drops even more significantly than for propositional logic when applying the standard positional encoding, namely to only 23.6% and 40.5%, respectively. With a tree positional encoding, however, the model preserves the semantic generalization. It outputs exact syntactic matches in 67.6% of the cases and achieves an astonishing overall accuracy of 92.2%. The results of our experiments are shown in the *shaded* part of Figure 3. Note that both positional encodings work equally well when not considering larger formulas.

Automata-based generator timeout. While generating the data set LTL_{35}, we stored 201 LTL formulas for which our automaton-based generator timed out (120 s). The Transformer constructs correct traces in 99% of the cases, i.e., for 199 out of 201 formulas. To give the interested reader an intuition on the complexity of these formulas, we provide an example here. The automaton construction timed out, but the Transformer was capable of outputting a satisfiable trace.

$\bigcirc\neg\bigcirc(\neg b\,\mathcal{U}\,\neg(\mathit{true}\,\mathcal{U}(\neg\bigcirc\bigcirc\bigcirc\bigcirc\bigcirc\bigcirc(e\wedge\mathit{true}\,\mathcal{U}\bigcirc(\mathit{true}\,\mathcal{U}\,d\,\mathcal{U}\,e))\wedge c\,\mathcal{U}\,\neg(c\,\mathcal{U}\bigcirc c))))$
`1; 1; b & c; !c; !c; 1; 1; 1; !e; {1}`

6 Related Work

Closely related to our work is NeuroSAT [36], which is a (message passing) graph neural network [33, 26, 16, 51] for solving the propositional satisfiability problem. We apply a standard sequence-to-sequence model, which allows us to avoid transforming formulas into CNF. Our approach can therefore be applied to logics which do not have a CNF. Our paper focusses on generalization properties of Transformers rather than showing that we can predict the satisfiability of a formula. Note that the variable counts are not comparable as their formulas are in CNF and ours include nested and more involved operators. A simplified NeuroSAT architecture was trained for unsat-core predictions [35], improving the performance of, for example, Z3 [8] by 6%. Similar learning techniques have been used to learn better heuristics for 2QBF solvers [24].

In [11], the authors study the problem of logical entailment, i.e., whether a formula φ_1 entails another propositional formula φ_2 and present the PossibleWorldNet, which is a specific architecture that evaluates the two formulas under consideration in different "worlds". They also contribute a data set for evaluating models on this task. Note that entailment is a subproblem of satisfiability and can be expressed in our framework by translating $\varphi_1 \rightarrow \varphi_2$ into a partial satisfying assignment.

Deep learning has recently been proposed for automating mathematical reasoning in (interactive) theorem provers [14, 27, 4, 29, 25] and other mathematical domains [32, 34]. Transformers were used to solve differential equations [23].

Transformers have also been considered for the analysis of code [12]. Earlier works applying Transformers studied natural language-like prediction tasks, such as summarizing code [12] or variable naming and misuse [20]. Other works focused on recurrent neural networks or graph neural networks for code analysis, e.g. [30, 17, 6, 48, 1]. Another area in the intersection of formal methods and machine learning is the verification of neural networks [37, 38, 41, 15, 21, 9].

7 Conclusion

We have shown that Transformers can generalize to the semantics of propositional and linear-time temporal logic. We considered the problem of *translating* a logical formula into a satisfying assignment or a satisfying trace, respectively. We showed that the Transformer learns the semantics of the logics instead of the particularities of the data generators. Our best performing models showed a 38.4% gap, for propositional logic and 15.1% gap, for LTL, between the predictions that where exact syntactic matches and the predictions that where semantically correct; overall achieving over 96.5%, and 98.3% respectively, accuracy on our test sets. The Transformer preserves this generalization even when challenged with formulas almost twice as large as it saw during training. We showed that the key to this generalization lies in a tree positional encoding. To our surprise, the Transformer also solved 199 out of 201 of the very hard LTL formulas on which even our automaton-based generator timed out.

The potential that arises from the advent of deep learning in logical reasoning is immense. Deep learning holds the promise to empower researchers in the automated reasoning and formal methods communities to abstract from minor implementation details and make bigger jumps in the development of new automated verification methods. The approach investigated in this paper could form the basis for hybrid algorithms, which would combine deductive and combinatorial approaches, as used traditionally in the formal methods and automated reasoning communities, with approaches inspired by the ongoing successes in deep learning.

Broader Impact

In their classical paper "On the Unusual Effectiveness of Logic in Computer Science" [18], Halpern, Harper, Immerman, Kolaitis, Vardi, and Vianu point out the crucial role of logical reasoning in nearly every aspect of information processing. It is hard to overstate the potential impact of combining neural networks with logical reasoning. Faster logical reasoning leads to faster query evaluation in data bases, faster constraint solving in operations research, and both faster and more comprehensive program verification and synthesis.

An important concern is that logical reasoning is often used in applications where correctness is critically important, such as in formal methods for the design of safety-critical systems. Errors, which a machine learning model is bound to produce from time to time, are not acceptable in such applications. Predictions of the model must therefore be validated before they can be trusted. Fortunately, validating a solution is usually much simpler than actually solving a logical formula. For example, checking whether an LTL formula has a satisfying trace is PSPACE-complete; checking that a given trace satisfies the formula can be done in polynomial time. In the rare cases where the validation fails, the logical reasoning will need to fall back to classical algorithms.

As with any technique that improves automation, it is worth considering the potential downside of taking away certain tasks from humans. For example, in program verification, it has been argued that it is specifically the logical reasoning process that gives the developer deeper insights about the program, and, hence, improves the quality of the code [43]. Automating logical reasoning shifts the human contribution from the low-level details to the more abstract level of logical specifications. The resulting loss of detail in the human understanding may well be an inevitable consequence of the much-desired gain in efficiency.

Acknowledgements

This work was partially supported by the Collaborative Research Center "Foundations of Perspicuous Software Systems" (TRR 248, 389792660) and by the European Research Council (ERC) Grant OSARES (No. 683300). We thank Jesko Hecking-Harbusch and Niklas Metzger for their valuable feedback on an earlier version of this paper.

References

[1] Miltiadis Allamanis, Marc Brockschmidt, and Mahmoud Khademi. Learning to represent programs with graphs. *arXiv preprint arXiv:1711.00740*, 2017.

[2] Gilles Audemard and Laurent Simon. On the glucose SAT solver. *Int. J. Artif. Intell. Tools*, 27(1): 1840001:1–1840001:25, 2018. doi: 10.1142/S0218213018400018. URL `https://doi.org/10.1142/S0218213018400018`.

[3] Mislav Balunovic, Pavol Bielik, and Martin Vechev. Learning to solve smt formulas. In S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett, editors, *Advances in Neural*

Information Processing Systems 31, pages 10317–10328. Curran Associates, Inc., 2018. URL http://papers.nips.cc/paper/8233-learning-to-solve-smt-formulas.pdf.

[4] Kshitij Bansal, Sarah M Loos, Markus N Rabe, Christian Szegedy, and Stewart Wilcox. HOList: An environment for machine learning of higher-order theorem proving. In *arXiv preprint arXiv:1904.03241*, 2019.

[5] Clark W. Barrett, Christopher L. Conway, Morgan Deters, Liana Hadarean, Dejan Jovanovic, Tim King, Andrew Reynolds, and Cesare Tinelli. CVC4. In Ganesh Gopalakrishnan and Shaz Qadeer, editors, *Computer Aided Verification - 23rd International Conference, CAV 2011, Snowbird, UT, USA, July 14-20, 2011. Proceedings*, volume 6806 of *Lecture Notes in Computer Science*, pages 171–177. Springer, 2011. doi: 10.1007/978-3-642-22110-1_14. URL https://doi.org/10.1007/978-3-642-22110-1_14.

[6] S. Bhatia, P. Kohli, and R. Singh. Neuro-symbolic program corrector for introductory programming assignments. In *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*, pages 60–70, May 2018. doi: 10.1145/3180155.3180219.

[7] Armin Biere, Alessandro Cimatti, Edmund M. Clarke, Ofer Strichman, and Yunshan Zhu. Bounded model checking. *Adv. Comput.*, 58:117–148, 2003. doi: 10.1016/S0065-2458(03)58003-2. URL https://doi.org/10.1016/S0065-2458(03)58003-2.

[8] Leonardo Mendonça de Moura and Nikolaj Bjørner. Z3: an efficient SMT solver. In C. R. Ramakrishnan and Jakob Rehof, editors, *Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings*, volume 4963 of *Lecture Notes in Computer Science*, pages 337–340. Springer, 2008. doi: 10.1007/978-3-540-78800-3_24. URL https://doi.org/10.1007/978-3-540-78800-3_24.

[9] Tommaso Dreossi, Alexandre Donzé, and Sanjit A Seshia. Compositional falsification of cyber-physical systems with machine learning components. *Journal of Automated Reasoning*, 63(4):1031–1053, 2019.

[10] Alexandre Duret-Lutz, Alexandre Lewkowicz, Amaury Fauchille, Thibaud Michaud, Etienne Renault, and Laurent Xu. Spot 2.0—a framework for ltl and ω-automata manipulation. In *International Symposium on Automated Technology for Verification and Analysis*, pages 122–129. Springer, 2016.

[11] Richard Evans, David Saxton, David Amos, Pushmeet Kohli, and Edward Grefenstette. Can neural networks understand logical entailment? *arXiv preprint arXiv:1802.08535*, 2018.

[12] Patrick Fernandes, Miltiadis Allamanis, and Marc Brockschmidt. Structured neural summarization. *arXiv preprint arXiv:1811.01824*, 2018.

[13] Bernd Finkbeiner and Sven Schewe. Bounded synthesis. *International Journal on Software Tools for Technology Transfer*, 15(5-6):519–539, 2013. ISSN 1433-2779. doi: 10.1007/s10009-012-0228-z. URL http://dx.doi.org/10.1007/s10009-012-0228-z.

[14] Thibault Gauthier, Cezary Kaliszyk, and Josef Urban. Tactictoe: Learning to reason with hol4 tactics. *arXiv preprint arXiv:1804.00595*, 2018.

[15] Timon Gehr, Matthew Mirman, Dana Drachsler-Cohen, Petar Tsankov, Swarat Chaudhuri, and Martin Vechev. Ai2: Safety and robustness certification of neural networks with abstract interpretation. In *2018 IEEE Symposium on Security and Privacy (SP)*, pages 3–18. IEEE, 2018.

[16] Justin Gilmer, Samuel S Schoenholz, Patrick F Riley, Oriol Vinyals, and George E Dahl. Neural message passing for quantum chemistry. In *Proceedings of the 34th International Conference on Machine Learning-Volume 70*, pages 1263–1272. JMLR. org, 2017.

[17] Rahul Gupta, Soham Pal, Aditya Kanade, and Shirish Shevade. Deepfix: Fixing common C language errors by deep learning. In *Thirty-First AAAI Conference on Artificial Intelligence*, 2017.

[18] Joseph Y. Halpern, Robert Harper, Neil Immerman, Phokion G. Kolaitis, Moshe Y. Vardi, and Victor Vianu. On the unusual effectiveness of logic in computer science. *Bulletin of Symbolic Logic*, 7(2):213–236, 2001. doi: 10.2307/2687775.

[19] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Delving deep into rectifiers: Surpassing human-level performance on imagenet classification. In *2015 IEEE International Conference on Computer Vision, ICCV 2015, Santiago, Chile, December 7-13, 2015*, pages 1026–1034, 2015. doi: 10.1109/ICCV.2015.123. URL https://doi.org/10.1109/ICCV.2015.123.

[20] Vincent J. Hellendoorn, Charles Sutton, Rishabh Singh, and Petros Maniatis. Global relational models of source code. In *International Conference on Learning Representations*, 2020. URL `https://openreview.net/forum?id=B1lnbRNtwr`.

[21] Xiaowei Huang, Marta Kwiatkowska, Sen Wang, and Min Wu. Safety verification of deep neural networks. In *International Conference on Computer Aided Verification*, pages 3–29. Springer, 2017.

[22] IEEE-Commission et al. Ieee standard for property specification language (psl). *IEEE Std 1850-2005*, 2005.

[23] Guillaume Lample and François Charton. Deep learning for symbolic mathematics. *arXiv preprint arXiv:1912.01412*, 2019.

[24] Gil Lederman, Markus N. Rabe, Edward A. Lee, and Sanjit A. Seshia. Learning heuristics for quantified boolean formulas through deep reinforcement learning. 2020. URL `http://arxiv.org/abs/1807.08058`.

[25] Dennis Lee, Christian Szegedy, Markus N. Rabe, Sarah M. Loos, and Kshitij Bansal. Mathematical reasoning in latent space. *CoRR*, abs/1909.11851, 2019. URL `http://arxiv.org/abs/1909.11851`.

[26] Yaguang Li, Rose Yu, Cyrus Shahabi, and Yan Liu. Diffusion convolutional recurrent neural network: Data-driven traffic forecasting. *arXiv preprint arXiv:1707.01926*, 2017.

[27] Sarah Loos, Geoffrey Irving, Christian Szegedy, and Cezary Kaliszyk. Deep network guided proof search. In *LPAR*, 2017.

[28] Matej Moravcík, Martin Schmid, Neil Burch, Viliam Lisý, Dustin Morrill, Nolan Bard, Trevor Davis, Kevin Waugh, Michael Johanson, and Michael H. Bowling. Deepstack: Expert-level artificial intelligence in no-limit poker. *CoRR*, abs/1701.01724, 2017. URL `http://arxiv.org/abs/1701.01724`.

[29] Aditya Paliwal, Sarah M. Loos, Markus N. Rabe, Kshitij Bansal, and Christian Szegedy. Graph representations for higher-order logic and theorem proving. *CoRR*, abs/1905.10006, 2019. URL `http://arxiv.org/abs/1905.10006`.

[30] Chris Piech, Jonathan Huang, Andy Nguyen, Mike Phulsuksombati, Mehran Sahami, and Leonidas Guibas. Learning program embeddings to propagate feedback on student code. In *International Conference on Machine Learning*, pages 1093–1102, 2015.

[31] Amir Pnueli. The temporal logic of programs. In *18th Annual Symposium on Foundations of Computer Science, Providence, Rhode Island, USA, 31 October - 1 November 1977*, pages 46–57, 1977. doi: 10.1109/SFCS.1977.32. URL `https://doi.org/10.1109/SFCS.1977.32`.

[32] David Saxton, Edward Grefenstette, Felix Hill, and Pushmeet Kohli. Analysing mathematical reasoning abilities of neural models. *CoRR*, abs/1904.01557, 2019. URL `http://arxiv.org/abs/1904.01557`.

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

[34] Imanol Schlag, Paul Smolensky, Roland Fernandez, Nebojsa Jojic, Jürgen Schmidhuber, and Jianfeng Gao. Enhancing the transformer with explicit relational encoding for math problem solving. *arXiv preprint arXiv:1910.06611*, 2019.

[35] Daniel Selsam and Nikolaj Bjørner. Guiding high-performance SAT solvers with unsat-core predictions. In *Theory and Applications of Satisfiability Testing - SAT 2019 - 22nd International Conference, SAT 2019, Lisbon, Portugal, July 9-12, 2019, Proceedings*, pages 336–353, 2019. doi: 10.1007/978-3-030-24258-9_24. URL `https://doi.org/10.1007/978-3-030-24258-9_24`.

[36] Daniel Selsam, Matthew Lamm, Benedikt Bünz, Percy Liang, Leonardo de Moura, and David L. Dill. Learning a SAT solver from single-bit supervision. In *7th International Conference on Learning Representations, ICLR 2019, New Orleans, LA, USA, May 6-9, 2019*, 2019. URL `https://openreview.net/forum?id=HJMC_iA5tm`.

[37] Sanjit A. Seshia and Dorsa Sadigh. Towards verified artificial intelligence. *CoRR*, abs/1606.08514, 2016. URL `http://arxiv.org/abs/1606.08514`.

[38] Sanjit A Seshia, Ankush Desai, Tommaso Dreossi, Daniel J Fremont, Shromona Ghosh, Edward Kim, Sumukh Shivakumar, Marcell Vazquez-Chanlatte, and Xiangyu Yue. Formal specification for deep neural networks. In *International Symposium on Automated Technology for Verification and Analysis*, pages 20–34. Springer, 2018.

[39] Vighnesh Leonardo Shiv and Chris Quirk. Novel positional encodings to enable tree-based transformers. In *NeurIPS 2019*, 2019. URL `https://www.microsoft.com/en-us/research/publication/novel-positional-encodings-to-enable-tree-based-transformers/`.

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

[41] Gagandeep Singh, Timon Gehr, Markus Püschel, and Martin Vechev. An abstract domain for certifying neural networks. *Proceedings of the ACM on Programming Languages*, 3(POPL):1–30, 2019.

[42] Alexander Smith, Andreas G. Veneris, Moayad Fahim Ali, and Anastasios Viglas. Fault diagnosis and logic debugging using boolean satisfiability. *IEEE Trans. on CAD of Integrated Circuits and Systems*, 24(10):1606–1621, 2005. doi: 10.1109/TCAD.2005.852031. URL `https://doi.org/10.1109/TCAD.2005.852031`.

[43] A. K. Sobel, H. Saiedian, A. Stavely, and P. Henderson. Teaching formal methods early in the software engineering curriculum. In *Software Engineering Education and Training, Conference on*, page 55, Los Alamitos, CA, USA, mar 2000. IEEE Computer Society. doi: 10.1109/CSEE.2000.827022. URL `https://doi.ieeecomputersociety.org/10.1109/CSEE.2000.827022`.

[44] Yaniv Taigman, Ming Yang, Marc'Aurelio Ranzato, and Lior Wolf. Deepface: Closing the gap to human-level performance in face verification. In *2014 IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2014, Columbus, OH, USA, June 23-28, 2014*, pages 1701–1708, 2014. doi: 10.1109/CVPR.2014.220. URL `https://doi.org/10.1109/CVPR.2014.220`.

[45] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need. In *Advances in Neural Information Processing Systems 30: Annual Conference on Neural Information Processing Systems 2017, 4-9 December 2017, Long Beach, CA, USA*, pages 5998–6008, 2017. URL `http://papers.nips.cc/paper/7181-attention-is-all-you-need`.

[46] Marcell Vazquez-Chanlatte. mvcisback/py-aiger, August 2018. URL `https://doi.org/10.5281/zenodo.1326224`.

[47] Jesse Vig. A multiscale visualization of attention in the transformer model. *arXiv preprint arXiv:1906.05714*, 2019. URL `https://arxiv.org/abs/1906.05714`.

[48] Ke Wang, Rishabh Singh, and Zhendong Su. Dynamic neural program embedding for program repair. In *ICLR*, 2018.

[49] Yonghui Wu, Mike Schuster, Zhifeng Chen, Quoc V. Le, Mohammad Norouzi, Wolfgang Macherey, Maxim Krikun, Yuan Cao, Qin Gao, Klaus Macherey, Jeff Klingner, Apurva Shah, Melvin Johnson, Xiaobing Liu, Lukasz Kaiser, Stephan Gouws, Yoshikiyo Kato, Taku Kudo, Hideto Kazawa, Keith Stevens, George Kurian, Nishant Patil, Wei Wang, Cliff Young, Jason Smith, Jason Riesa, Alex Rudnick, Oriol Vinyals, Greg Corrado, Macduff Hughes, and Jeffrey Dean. Google's neural machine translation system: Bridging the gap between human and machine translation. *CoRR*, abs/1609.08144, 2016. URL `http://arxiv.org/abs/1609.08144`.

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

[51] Zonghan Wu, Shirui Pan, Fengwen Chen, Guodong Long, Chengqi Zhang, and Philip S Yu. A comprehensive survey on graph neural networks. *arXiv preprint arXiv:1901.00596*, 2019.

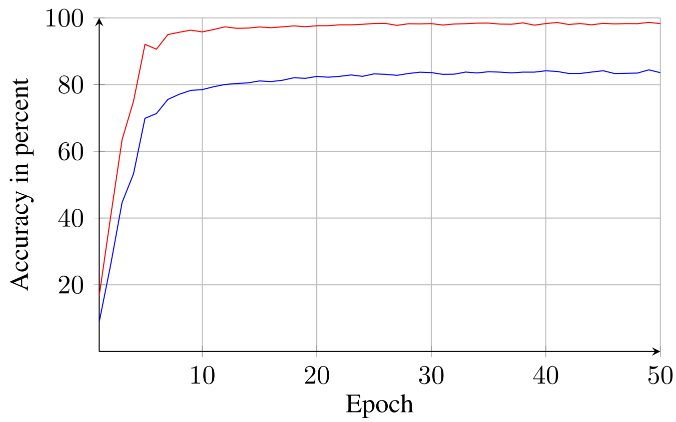


Figure 4: Exact syntactic match accuracy (blue) and total accuracy (red) of our best performing model, evaluated on a subset of 5K samples of LTL_{35} per epoch.

A LTL Semantics

The formal syntax of LTL is given by the following grammar:

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

where $p \in AP$ is an atomic proposition. Let AP be a set of *atomic propositions*. A (*explicit*) *trace* t is an infinite sequence over subsets of the atomic propositions. We define the set of traces $TR := (2^{AP})^{\omega}$. We use the following notation to manipulate traces: Let $t \in TR$ be a trace and $i \in \mathbb{N}$ be a natural number. With $t[i]$ we denote the set of propositions at i-th position of t. Therefore, $t[0]$ represents the starting element of the trace. Let $j \in \mathbb{N}$ and $j \geq i$. Then $t[i,j]$ denotes the sequence $t[i]\ t[i+1] \ldots t[j-1]\ t[j]$ and $t[i,\infty]$ denotes the infinite suffix of t starting at position i.

Let $p \in AP$ and $t \in TR$. The semantics of an LTL formula is defined as the smallest relation $\models$ that satisfies the following conditions:

$t \models p$	iff	$p \in t[0]$
$t \models \neg\varphi$	iff	$t \not\models \varphi$
$t \models \varphi_1 \wedge \varphi_2$	iff	$t \models \varphi_1$ and $t \models \varphi_2$
$t \models \bigcirc\varphi$	iff	$t[1,\infty] \models \varphi$
$t \models \varphi_1\,\mathcal{U}\,\varphi_2$	iff	there exists $i \geq 0 : t[i,\infty] \models \varphi_2$
		and for all $0 \leq j < i$ we have $t[j,\infty] \models \varphi_1$

B Accuracy During Training

In Figure 4 we show the evolution of both the exact syntactic matches and the semantic accuracy during the training process. Note the significant difference between the exact syntactic matches and the total accuracy right from the beginning. This demonstrates the importance of a suitable performance measure when evaluating machine learning algorithms on logical reasoning tasks.

C Handcrafted Examples

Example Predictions To evaluate and inspect the results, we ran the Transformer on several handcrafted examples. The Transformer has never seen these example inputs during training. The evaluation on our handcrafted examples examines to what extend the training on random input formulas transfers to "typical" LTL formulas.

The first formula combines the temporal modalities "globally" and "eventually", which is a common pattern in specifications for reactive systems. It requires that the atomic proposition a appears infinitely often on a trace. The Transformer outputs a trace with an empty prefix and a period containing a, i.e., every trace that contains a infinitely often.

$\square\Diamond a$	a^{ω}
`!U1!U1a`	`{a}`

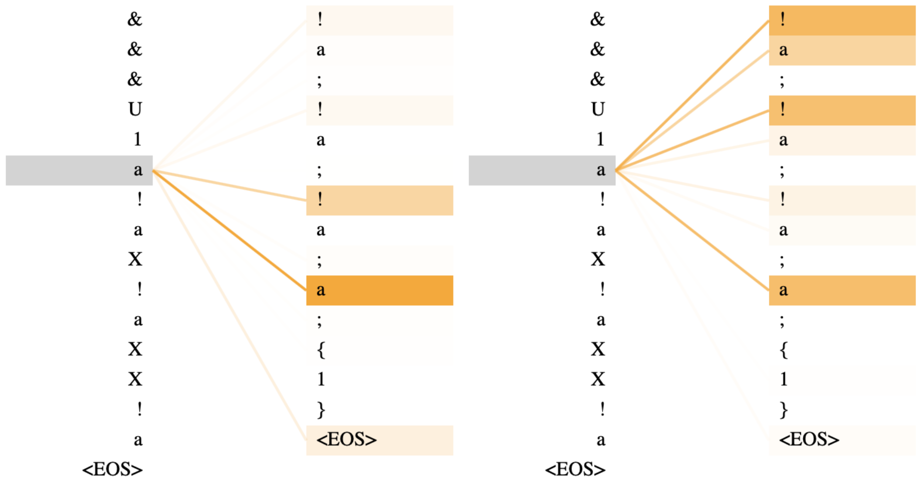


Figure 5: Two Encoder-decoder attention heads between the formula $\Diamond a \wedge \neg a \wedge \bigcirc \neg a \wedge \bigcirc\bigcirc \neg a$ and the output trace $\neg a\ ;\ \neg a\ ;\ \neg a\ ;\ a\ ;\ \mathit{true}^{\omega}$.

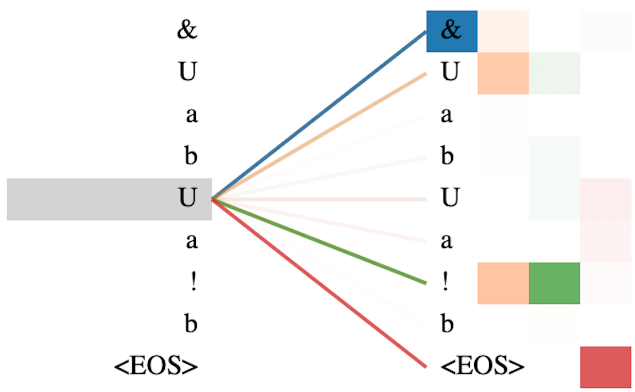


Figure 6: All self-attention heads of the formula $a\,\mathcal{U}\,b \wedge a\,\mathcal{U}\,\neg b$. Each color corresponds to a different attention head.

The second example formula requires that the atomic proposition a has to hold eventually, but not at the first three positions of the trace. The Transformer constructs a trace where a is not allowed to hold on the first three positions and fulfills the $\Diamond a$ requirement directly at the fourth position, before allowing an arbitrary period.

$\Diamond a \wedge \neg a \wedge \bigcirc \neg a \wedge \bigcirc\bigcirc \neg a$	$(\neg a)\ (\neg a)\ (\neg a)\ a\ (\mathit{true})^{\omega}$
`&&&U1a!aX!aXX!a`	`!a;!a;!a;a;{1}`

For this example, we visualized the attention mechanism in Fig. 5. When trying to satisfy $\Diamond a$, both heads pay close attention to the negation of the first three positions, which would lead to a contradiction if the Transformer decides to place an a before the fourth position.

The third example shows that the Transformer avoids such contradictions even in association with temporal operators. The formula requires that eventually an a has to hold as well as a position where a is not allowed to hold. The Transformer avoids a contradiction by first fulfilling the first conjunct $\Diamond a$ on the first position and then the second conjunct $\Diamond \neg a$ on the second position.

$\Diamond a \wedge \Diamond \neg a$	$a\ (\neg a)\ (\mathit{true})^{\omega}$
`&U1aU1!a`	`a;!a;{1}`

Example of a Misprediction Our last two examples describe formulas with multiple until statements that describe overlapping intervals. We know that these formulas are hard as they are the source of PSPACE-hardness of LTL.

The first formula overlaps two until intervals by requiring that a has to hold until b holds, as well as $\neg b$ holds. Here, the Transformer still predicts a correct trace: The predicted trace first satisfies $\neg b$ at the first position while delaying the satisfaction of the first until to the second position by requiring a at the first position as well. Figure 6 shows the self-attention heads for this formula. While processing the second until operator, the blue attention head pays attention to its top level operator, the conjunction.

$a \mathcal{U} b \wedge a \mathcal{U} \neg b$	$(a \wedge \neg b)\ b\ (true)^{\omega}$
`&UabUa!b`	`&a!b;b;{1}`

We scaled this formula to three overlapping until intervals, and observe that the Transformer fails: It predicts the trace $a \wedge \neg b\ ;\ b \wedge c;\ true^{\omega}$, which does not satisfy the LTL formula.

$(a \mathcal{U} b \wedge c) \wedge (a \mathcal{U} \neg b \wedge c) \wedge (a \mathcal{U} b \wedge \neg c)$	$(a \wedge \neg b)\ (b \wedge c)\ (true)^{\omega}$
`&&Ua&bcUa&!bcUa&b!c`	`&a!b;&bc;{1}`

Since the Transformer can solve this hard logical reasoning task for two until statements, we expect it to scale very well when being trained with more GPUs on larger formulas. Especially because the architecture of the Transformer allows for heavy parallelization of the learning task.