Learning Finite Linear Temporal Logic Specifications with a Specialized Neural Operator

Anonymous Authors[1]

Abstract

Finite linear temporal logic (LTL_f) is a powerful formal representation for modeling temporal sequences. We address the problem of learning a compact LTL_f formula from labeled traces of system behavior. We propose a novel neural network operator and evaluate the resulting architecture, DeepLTL_f. Our approach includes a specialized recurrent filter, designed to subsume LTL_f temporal operators, to learn a highly accurate classifier for traces. Then, it discretizes the activations and extracts the truth table represented by the learned weights. This truth table is converted to symbolic form and returned as the learned formula. Experiments on randomly generated LTL_f formulas show DeepLTL_f scales to larger formula sizes than existing approaches and maintains high accuracy even in the presence of noise.

1. Introduction

Recurrent neural networks (RNNs) have proven highly effective at learning classifiers for sequential data. Yet, RNNs typically employ a large number of parameters leading to a lack of interpretability in the decisions they make. Linear temporal logic (LTL) and its finite variant (LTL_f) are alternative representations for classifying sequential data in a human-understandable manner (Pnueli, 1977; De Giacomo & Vardi, 2013). However, learning LTL_f formulas has proven to be a difficult task. We propose DeepLTL_f, a new technique for learning classifiers for temporal behavior that combines the ease of optimization of RNNs with the interpretability of LTL_f.

LTL_f learning techniques can produce formulas for LTL_f synthesis, or the automated construction of a program according to formal specifications (Pnueli & Rosner, 1989). LTL_f is especially applicable in learning from demonstrations. A human teacher demonstrates the desired behavior to a learning agent, the agent produces an LTL_f formula summarizing the behavior (Vazquez-Chanlatte et al., 2018; Kasenberg & Scheutz, 2017), and the formula is used in place of a reward function in the context of reinforcement learning (Littman et al., 2017; Li et al., 2017).

We examine the problem of producing a compact LTL_f formula that correctly classifies traces of system behavior given labeled examples.

Definition 1 (LTL_f Learning Problem)**.** *Given a set of finite length positive traces, Π_P, and a set of finite length negative traces, Π_N, produce a compact LTL_f formula satisfied by the positive traces and violated by the negative traces.*

The LTL_f learning approach of Camacho & McIlraith (2019) is most directly related to our work. They reduce the LTL_f learning problem to SAT, but their method does not scale well to larger formula sizes or trace sets. Unlike DeepLTL_f, their method fails to find an appropriate formula when the data is noisy and not perfectly separable with a formula of the specified size. Neider & Gavran (2018) combine SAT solving and decision trees to produce LTL formulas in the presence of noisy data, but they face similar scaling issues. Kim et al. (2019) use Bayesian inference to learn LTL formulas for a limited set of LTL templates.

Our contributions are:

1. DeepLTL_f, a method for producing LTL_f formulas that classify traces.
2. Evaluation of DeepLTL_f on synthetic data for qualitative LTL_f formulas.
3. Comparison of DeepLTL_f to SAT-based approaches.

2. Linear Temporal Logic

Linear temporal logic (LTL) is a formal language used to express temporal properties of sequential data. LTL formulas consist of a set of propositions $p \in P$, standard logical operators, and temporal operators. Formulas are evaluated over traces, π, which are sequences of truth assignments to all the propositions in P. LTL is defined on infinite-length traces. In this paper, we use a variant, LTL_f, defined on

[1]Anonymous Institution, Anonymous City, Anonymous Region, Anonymous Country. Correspondence to: Anonymous Author <anon.email@domain.com>.

Preliminary work. Under review by the International Conference on Machine Learning (ICML). Do not distribute.

finite-length traces (De Giacomo & Vardi, 2013). The notation $\pi, t \models \phi$ denotes that the formula ϕ holds at timestep t in trace π where $0 \leq t < T$, and T is the trace length. When $\pi, 0 \models \phi$, we say the trace π *satisfies* the formula ϕ.

The minimal temporal operators are next (X) and until (U). Next, $\mathsf{X}\,\phi$, denotes that ϕ will hold in the following timestep, while until, $\phi \,\mathsf{U}\, \psi$, denotes that ϕ must hold until ψ becomes true. A number of temporal operators can be formed from these operators. The eventually (F) operator denotes that a variable holds at some timestep in the future: $\mathsf{F}\,\phi \iff \mathit{true}\,\mathsf{U}\,\phi$. The globally ($\mathsf{G}$) operator denotes that a variable holds at all subsequent timesteps: $\mathsf{G}\,\phi \iff \neg\,\mathsf{F}\,\neg\phi$. LTL_f introduces an additional temporal operator, weak next (N), to address behavior at the end of a trace. Weak next, $\mathsf{N}\,\phi$, denotes that ϕ must hold at the next time step *or* the next time step does not exist. In LTL_f, $\neg\,\mathsf{X}\,\mathit{true}$ holds at only the last timestep. In this paper, we also use the weak until operator defined as $\phi\,\mathsf{W}\,\psi \iff (\phi\,\mathsf{U}\,\psi) \vee \mathsf{G}\,\phi$. Weak until is similar to $\phi\,\mathsf{U}\,\psi$, except that ψ does not need to occur. Finally, the fragment of LTL consisting of only next and next-derived operators is called *metric* LTL, while the fragment consisting of only until and until-derived operators is called *qualitative* LTL.

3. DeepLTL_f

Inspired by the temporal operators, we present a novel network architecture for classifying traces. Layers in the network consist of multiple filters, similar to convolutional filters (Fukushima, 1979). The filters in each layer are “soft” versions of LTL_f operators. By stacking layers in the network, subsequent filters are applied to the results of previous filters, equivalent to the nesting of LTL_f operators in a formula. The entire network encodes a single LTL_f formula.

The first layer of our network takes as input traces from the positive and negative trace sets. The truth values of the propositions in the traces are interpreted as 1 and 0 for *true* and *false*, respectively. The filters in the first layer are applied to the trace to generate a sequence of activations in $[0, 1]$. These activations form a new trace, which becomes the input to the subsequent layer. These intermediate traces represent the truth values of the soft LTL_f operators encoded by the filters. The activations of the final layer correspond to the network's prediction of the truth value for every timestep in the original trace. Corresponding to the semantics of LTL_f, we use the truth value of the first timestep in the network output as the predicted label of the trace. We compare the predicted and target labels of the trace to compute a loss that we minimize via gradient descent. Figure 1 depicts a DeepLTL_f network.

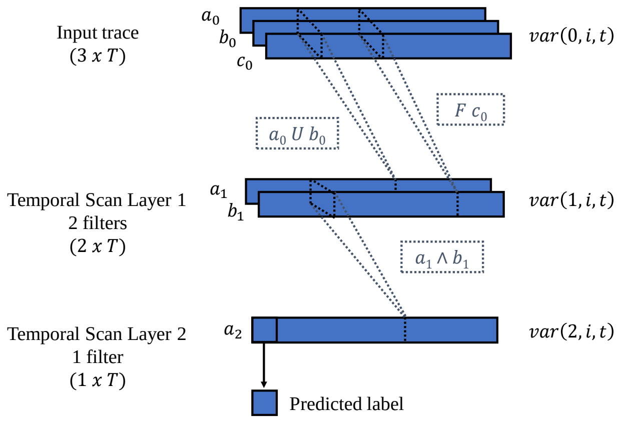

Figure 1. A DeepLTL_f network that encodes the formula $(a\,\mathsf{U}\,b) \wedge \mathsf{F}\,c$. Solid boxes are the input trace and output activations. Dashed lines represent the application of the filters. The formulas in the dashed boxes represent formula fragments learned by each filter.

3.1. Network Weights

Each layer, l, of the network consists of at least 1, but possibly multiple, filters indexed by i. These filters act on a sequence of truth values for variables indexed by j. The sequence of truth values might be the original trace, in the case of the first layer, or the output of the previous layer. We use $\mathrm{var}(l, i, t)$ to denote the activation of filter i at timestep t in layer l. The input trace is $\mathrm{var}(0, i, t)$.

A filter consists of a set of weights that allow for the expression of standard logical operators and temporal operators.

- $W_P(l, i, j)$ is the propositional weight of filter i in layer l for variable j and allows for the expression of standard logical operators.
- $W_M(l, i, j)$ is the metric weight of filter i in layer l for variable j and allows for the expression of metric temporal operators.
- $W_Q(l, i)$ is the qualitative weight of filter i in layer l and allows for the expression of qualitative temporal operators.
- $b(l, i)$ is the bias term for filter i in layer l.
- $\mathrm{var}(l-1, j, T+1)$ and $\mathrm{var}(l, i, T+1)$ are base values.

Together, these weights define a linear classifier that gives the truth value of the soft LTL_f operator represented by the filter. The weights of one DeepLTL_f filter can represent various LTL_f operators (Table 1).

We apply a filter to a sequence using the following formula:

$$\begin{aligned}\mathrm{var}(l,i,t) = \sigma\Bigg(&\sum_j W_P(l,i,j)\mathrm{var}(l-1,j,t) \;+ \\ &\sum_j W_M(l,i,j)\mathrm{var}(l-1,j,t+1) \;+ \\ &\delta(W_Q(l,i))\mathrm{var}(l,i,t+1) + b(l,i)\Bigg),\end{aligned} \tag{1}$$

where δ is the function $\max(0,x)$ since we require that W_Q is positive for formula extraction (Section 3.2). However, δ takes a slightly different form for training (Section 3.4). Similarly, σ is the binary step function, $\mathbb{1}_{[0,\infty)}$, for formula extraction and sigmoid activation during training.

In words, applying a filter to a sequence is a recursive operation in the timestep t, corresponding to the recursive evaluation of an LTL_f operator on a trace. The recursion begins at timestep T and the base case values, $\mathrm{var}(l-1,j,T+1)$ and $\mathrm{var}(l,i,T+1)$, are parameters learned along with the weights. Running backwards, the output of a filter is computed as σ applied to the sum of the propositional weights applied to the variables at the current timestep, the metric weights applied to the variables at the next timestep, and the qualitative weight applied to the output of the filter at the next timestep.

3.2. Conversion from Network Weights to Formula

After a network has been trained, the learned weights can be interpreted as an LTL_f formula. In this interpretation, each DeepLTL$_f$ filter encodes an LTL_f expression that has the form $\phi \,\mathsf{U}\, \psi$ or $\phi \,\mathsf{W}\, \psi$. Here, ϕ, ψ are Boolean expressions in full disjunctive normal form (fDNF) with additional literals for the next state of each proposition in each clause that are prepended by X or N operators. This space of augmented DNF expressions joined by U or W will be referred to as *temporal normal form* (TNF) expressions.

To facilitate the interpretation of DeepLTL$_f$ filters, we first define a concept we call a *temporal truth table*. A temporal truth table is similar to a standard Boolean logic truth table, but is augmented with extra information specific to DeepLTL$_f$ filters. It encodes an LTL_f expression in temporal normal form using the usual columns for each of the n propositions x_j, and the output column f. In addition, for each x_j, temporal truth tables have a corresponding column m_j, collectively called the metric bits. These bits encode the semantics of the metric operators X and N by representing the value of each proposition one timestep into the future. Additionally, there is a column τ, the temporal bit, that represents the future truth value of the formula. It encodes the semantics of the qualitative operators U, W, F and G since their truth values depend on future values in the trace. Separate from the columns, each temporal truth table has an additional $n+1$ bits of information that encode the filter's behavior at the end of the trace. The first of these bits, Ω, is calculated by passing the filter's learned base case value $\mathrm{var}(l,i,T+1)$ through the binary step function. The other n bits, ω_j, are found by passing the learned base case values $\mathrm{var}(l-1,j,T+1)$ through the binary step function.

The rows for the temporal truth table are filled by applying the values for each of the bits in the truth table to the appropriate input of the filter. The x_j columns are multiplied by the W_P weights, the m_j columns are multiplied by the W_M weights, and the τ column is multiplied by the W_Q weight. The process of converting a filter to a temporal truth table discretizes the continuous operation of the filter. The conversion algorithm is outlined in Algorithm 1. An example completed temporal truth table is shown in Table 2.

Algorithm 1 Convert Filter to Temporal Truth Table

```
Input: filter layer $l$, filter index $i$, trace length $T$, number of variables $n$
$f \leftarrow$ empty truth table
$\Omega \leftarrow \sigma(\mathrm{var}(l,i,T+1))$
for $j \in \{1 \ldots n\}$ do
    $\omega_j \leftarrow \sigma(\mathrm{var}(l,j,T+1))$
end for
for $k \in \{0,1\}^{2n+1}$ do
    $x_1, x_2, \ldots, x_n, m_1, m_2, \ldots, m_n, \tau \leftarrow k$
    $f[k] \leftarrow \sigma(\Sigma_j W_P(l,i,j)x_j + \Sigma_j W_M(l,i,j)m_j + \delta(W_Q(l,i))\tau + b(l,i))$
end for
return $f$, $\Omega$, $\omega$
```

Interpreting the temporal truth table is straightforward because the table can be used to construct a formula in temporal normal form. The operator is determined by Ω, since whether an until operator is weak or strong is determined by the base case values: U for $\Omega = 0$ and W for $\Omega = 1$. The formula ϕ is created by taking the disjunction of the conjunction (Rautenberg, 2010) of all the proposition and metric bits when $f = 1$ and $\tau = 1$. Formula ψ is created by taking the disjunction of the conjunction of the proposition and metric bits in the rows where $f = 1$ and $\tau = 0$. The metric bits are prepended with X when the corresponding $\omega = 0$ and with N when the corresponding $\omega = 1$, since the choice of X or N is determined by base case values. The rows in which $f = 0$ do not contribute to the expression's representation. Algorithm 2 outlines the procedure of converting a temporal truth table to a formula.

3.3. Correctness of Conversion Procedure

Because temporal truth tables represent TNF expressions, there are some settings of the table that result in logically impossible expressions, and are therefore invalid. Specifically, for any given setting of the x_j, m_j bits, it cannot

Table 1. Example weights for LTL_f operators: Assume one DeepLTL$_f$ filter, i, applied to two truth value sequences representing the LTL_f formulas ϕ and ψ. Standard logical operators are also easy to express using the W_P weights. Weights that can take any value are marked with $-$. We abuse notation and use $W(l, i, \phi)$ to mean the weight applied to the truth value sequence representing ϕ.

LTL_f Op.	$W_P(l,i,\phi)$	$W_P(l,i,\psi)$	$W_M(l,i,\phi)$	$W_M(l,i,\psi)$	$W_Q(l,i)$	$b(l,i)$	$\text{var}(l-1,\phi,T+1)$	$\text{var}(l,i,T+1)$
$\phi\,\mathsf{U}\,\psi$	1	2	0	0	1	-1.5	$-$	0
$\phi\,\mathsf{W}\,\psi$	1	2	0	0	1	-1.5	$-$	1
$\mathsf{X}\,\phi$	0	0	1	0	0	-0.5	0	$-$
$\mathsf{N}\,\phi$	0	0	1	0	0	-0.5	1	$-$
$\mathsf{F}\,\phi$	1	0	0	0	1	-0.5	$-$	0
$\mathsf{G}\,\phi$	1	0	0	0	1	-1.5	$-$	1

Algorithm 2 Convert Temporal Truth Table to Formula

```
Input: number of vars $n$, temporal truth table $f$, $\Omega$, $\omega$
$\phi \leftarrow$ False
$\psi \leftarrow$ False
for $k \in \{0,1\}^{2n+1}$ do
  $x_1, x_2, \ldots, x_n, m_1, m_2, \ldots, m_n, \tau \leftarrow k$
  if $f[k] = 1$ then
    $c \leftarrow$ True
    for $j \in \{1 \ldots n\}$ do
      if $x_j = 1$ then
        $b_j \leftarrow x_j$
      else
        $b_j \leftarrow \neg x_j$
      end if
      if $m_j = 1$ then
        $d_j \leftarrow x_j$
      else
        $d_j \leftarrow \neg x_j$
      end if
      if $\omega_j = 1$ then
        $c \leftarrow c \wedge b_j \wedge \mathsf{N}\, d_j$
      else
        $c \leftarrow c \wedge b_j \wedge \mathsf{X}\, d_j$
      end if
    end for
    if $\tau = 1$ then
      $\phi \leftarrow \phi \vee c$
    else
      $\psi \leftarrow \psi \vee c$
    end if
  end if
end for
if $\Omega = 1$ then
  return $\phi\,\mathsf{W}\,\psi$
else
  return $\phi\,\mathsf{U}\,\psi$
end if
```

be the case that the row with $\tau = 0$ has $f = 1$ and the corresponding row with $\tau = 1$ has $f = 0$. If this were the case, that would mean that some clause of the temporal expression appears in ψ, but explicitly does not appear in ϕ. This situation cannot occur because a clause's existence in ψ guarantees that it is implicitly in ϕ, by definition of the until operation. Tables that have such a property that create logically impossible expressions will be referred to as *invalid*. By design, DeepLTL$_f$ filters create only valid truth tables when trained.

Lemma 1. *Any Deep*LTL_f *filter will produce a valid temporal truth table.*

Proof. An invalid truth table results when some setting of the x_j and m_j bits produces an output of $f = 0$ when $\tau = 1$ and $f = 1$ when $\tau = 0$. We assume for contradiction that we have an invalid truth table.

Using Equation 1, τ from the truth table is represented by $\text{var}(l, i, t+1)$ and the filter activation, f, is $\text{var}(l, i, t)$. Consider some filter i applied to identical settings of the propositional variables, but when $\text{var}(l, i, t+1) = 1$ then $\text{var}(l, i, t) = 0$, and when $\text{var}(l, i, t'+1) = 0$ then $\text{var}(l, i, t') = 1$. This situation is precisely what would cause the filter to produce an invalid table. Note that

$$\begin{aligned} \text{var}(l,i,t) &< \text{var}(l,i,t') \\ \delta(W_Q(i))\text{var}(l,i,t+1) &< \delta(W_Q(i))\text{var}(l,i,t'+1) \\ \delta(W_Q(i)) &< 0. \end{aligned}$$

The second line is obtained by substitution from Equation 1. The derivation shows that, for a filter to create an invalid table, $\delta(W_Q(i)) < 0$. However, $\delta = \max(0, x)$, so $\delta(W_Q(i))$ is non-negative for any DeepLTL$_f$ filter. Therefore, all DeepLTL$_f$ filters produce valid truth tables. □

Beyond the filters encoding only valid truth tables, it is important that the method for interpreting those tables from

Table 2. Temporal truth table for the formula $x_1 \mathsf{U} x_2$. The rows in which $f = 0$ do not contribute and were omitted for space. The values of the literals from this table are applied to the weights of the learned filter, shown in the $\phi \mathsf{U} \psi$ row of Table 1 with $\phi = x_1$ and $\psi = x_2$. The resultant value determines f. Here, Ω, ω_1, and ω_2 are calculated by the step function applied to $\mathrm{var}(l, i, T+1)$, $\mathrm{var}(l-1, x_1, T+1)$ and $\mathrm{var}(l-1, x_2, T+1)$, respectively.

$\Omega = 0$		$\omega_1 = 0$		$\omega_2 = 0$	
x_1	x_2	m_1	m_2	τ	f
0	1	0	0	0	1
0	1	0	0	1	1
0	1	0	1	0	1
0	1	0	1	1	1
0	1	1	0	0	1
0	1	1	0	1	1
0	1	1	1	0	1
0	1	1	1	1	1
1	0	0	0	1	1
1	0	0	1	1	1
1	0	1	0	1	1
1	0	1	1	1	1
1	1	0	0	0	1
1	1	0	0	1	1
1	1	0	1	0	1
1	1	0	1	1	1
1	1	1	0	0	1
1	1	1	0	1	1
1	1	1	1	0	1
1	1	1	1	1	1

and into LTL_f formulas is correct. Specifically, it should be the case that DeepLTL_f filters can successfully be interpreted into LTL_f expressions. We note here that a successful interpretation is one that results in a valid expression that approximates, but need not exactly match the behavior of, the DeepLTL_f filter—that happens since the interpretation uses a binary step function to discretize the operation of the filter. We also require that a successful interpretation create a temporal truth table that is LTL_f*-expression preserving*. That is, any temporal truth table created from a given LTL_f expression will result in an equivalent LTL_f expression.

Theorem 1. *Given a learned Deep*LTL_f *filter, the process of interpreting its weights into* LTL_f *expressions is correct—Deep*LTL_f *filters encode valid temporal truth tables that are* LTL_f*-expression preserving. That is, given an* LTL_f *expression g and its temporal truth table T, one can create a formula h from T via Algorithm 2. Then, $g = h$ and T is a valid temporal truth table.*

Proof. By Lemma 1, any temporal truth tables created by a DeepLTL_f filter are valid. To prove the soundness of our interpretation method, we must show that any valid temporal truth table is LTL_f-expression preserving.

Consider an arbitrary LTL_f expression g, in TNF. Evaluating this expression for every assignment of the variables in the temporal truth table will allow us to construct a valid temporal truth table. Algorithm 2 creates a TNF expression from the table, h. Assume for contradiction that g and h differ in some way. For h to differ from g, it must be missing a clause, have an additional clause in ϕ or ψ, or have a different operator than g.

If h is missing a clause that was in g, that implies that the value of the temporal truth table for that clause was 0. However, if that clause was in g, then its value in the table would have been 1—a contradiction.

If h has an extra clause that g does not have, the value of that clause in the table was 1. However, if that clause was not present in g, then the corresponding value of the table for that row would be 0—a contradiction.

For h to have a U where g has a W or an X where g has a N or vice versa, it would need to have a 0 where g has a 1 or vice versa in the $n + 1$ extra bits of the truth table. That would contradict that the truth table was computed from g.

Any difference in g and h results in a contradiction in the structure of the temporal truth table, therefore g and h are identical. This argument shows that our method of interpreting truth tables is LTL_f-expression preserving. Since DeepLTL_f filters encode only valid tables (Lemma 1), and our method for interpreting valid tables is sound, we can interpret any DeepLTL_f filter as a valid LTL_f expression. □

The ability to train and interpret individual DeepLTL_f filters as LTL_f expressions enables the construction of complex LTL_f formulas from deep networks using DeepLTL_f filters. The structure of the network dictates the composition of the interpreted expressions to create the overall formula that was learned by the network.

3.4. Implementation Details

We implemented DeepLTL_f in Tensorflow using binary cross-entropy loss optimized with Adam (Kingma & Ba, 2015). A formula that is satisfied by all the positive traces and violated by all the negative traces will have the minimum cross-entropy loss since the DeepLTL_f network that encodes the formula will perfectly classify every trace. We employ several procedures that increase the accuracy and compactness of the formulas output by DeepLTL_f.

Logic Minimization While every DeepLTL_f filter can be converted into a TNF formula, the TNF formula is generally not human-readable due to its large size. Rather than returning a TNF formula directly, we use the Espresso logic-minimization algorithm to initially reduce the formula from the filter's temporal truth table into a more compact formula (Rudell, 1986). Then, we use the Spot LTL_f library to further reduce the formula according to LTL_f simplification rules (Duret-Lutz et al., 2016). Although Spot is designed for LTL rather than LTL_f, we prove in the Supplementary Material that our use of Spot is valid for LTL_f. On average, in our experiments, Espresso reduced the size of the initial TNF formula (3278367, on average) by 91% (to 650, on average) and Spot by a further 51% (to 17, on average).

Annealing and Random Restarts There is inevitable information loss when converting the continuous network weights into a discrete temporal truth table. However, to encourage DeepLTL_f to learn representations that maintain high accuracy when discretized, we linearly increase the steepness of the sigmoid activation, σ, as training progresses. Similarly, while we use the δ function to restrict W_Q to positive values at test time (see Section 3.1), we relax this restriction during training. We define a "leaky" δ with a small positive slope in the negative region. This negative-region slope is linearly reduced as training progresses. Specifically, given the definitions of σ and δ parameterized by β and α:

$$\sigma(x) = \frac{1}{1+e^{-\beta x}} \quad \text{and} \quad \delta(x) = \max(x, \alpha x).$$

We use annealing rates α_d and β_d, updating the values of α and β at the end of each epoch by setting $\alpha = \alpha + \alpha_d$ and $\beta = \beta + \beta_d$.

Since formula extraction replaces the sigmoid activation with the binary step function and the leaky δ with the strict δ, annealing these activations during training increases the likelihood that the extracted formulas will match the behavior of the network. To further increase the chances of learning weights that discretize well, we also use random restarts. We train the network multiple times with different random weight initializations and use the trained network that has the highest accuracy after discretization. We attempted to employ L_1 regularization to the activations to further encourage better discretization, but found this made optimization too difficult for Adam.

Multiple Networks In principle, even a very large DeepLTL_f network can produce a compact formula after simplification. However, since larger networks can represent larger formulas, they pose a greater risk of producing a formula that overfits the training data. To balance the goal of learning a compact but also highly accurate formula, we train multiple networks each with a different number of filters on a given set of data. We then choose the smallest formula of the set of formulas with the highest accuracy after extraction.

4. Experiments

We evaluated our LTL_f formula learner, comparing it to approaches from the literature.

4.1. DeepLTL_f vs. SAT

To test the scalability of DeepLTL_f with respect to formula size, we evaluated its performance on data from random formulas. Then, using the same data, we swapped 1% of the labels to additionally test DeepLTL_f's ability to handle noise. For both experiments, we compared DeepLTL_f with the SAT-based approach by Camacho & McIlraith (2019) since their method does not make use of LTL_f templates to restrict the space of learnable formulas, like Kim et al. (2019), and works out-of-the-box with LTL_f rather than LTL, unlike Neider & Gavran (2018). We use their SAT encoding in conjunction with the associated learning algorithm. The algorithm iteratively increases the maximum allowed formula size and reruns the SAT solver until a formula is found. This process guarantees the output formula is optimally compact.

To increase robustness to noisy labels, we also devised a novel variant of the SAT approach. In the partial maximum satisfiability (PMAX-SAT) problem (Cha et al., 1997), rather than simply finding a satisfying truth assignment for a Boolean formula, the goal is to satisfy the *maximum* number of a designated set of "soft" clauses, while satisfying all of the remaining "hard" clauses. Our PMAX-SAT variant uses the same SAT encoding from Camacho & McIlraith (2019), but designates the clauses enforcing trace satisfaction as soft clauses. Thus, satisfying the maximum number of soft constraints in the PMAX-SAT problem corresponds to producing a formula satisfied by the maximum number of traces. Given the PMAX-SAT problem encoding, we execute a PMAX-SAT solver to learn a formula from the trace data. With the PMAX-SAT variant, we wanted to test whether a modified SAT-based approach could handle noise without prohibitively increasing runtime.

Data First, we generated random qualitative LTL_f formulas with $|P| = 3$ by uniform sampling of the LTL_f grammar. We generated 50 of each length ranging from 2 to 15 (or as many as possible if the number of unique formulas of a given size was less than 50). The length of an LTL_f formula is the sum of the number of temporal operators, binary logical operators, and propositions in the formula. We threw out formulas that did not include a temporal operator, meaning there were no formulas of size 1. We converted the formulas into negative normal form following the precedent set by Camacho & McIlraith (2019). Then, we adopted an approach from Camacho & McIlraith (2019) and generated a *characteristic sample* of traces for each formula's corresponding minimal deterministic finite-state automaton (DFA) (Parekh & Honavar, 2001). A set of labeled traces is considered

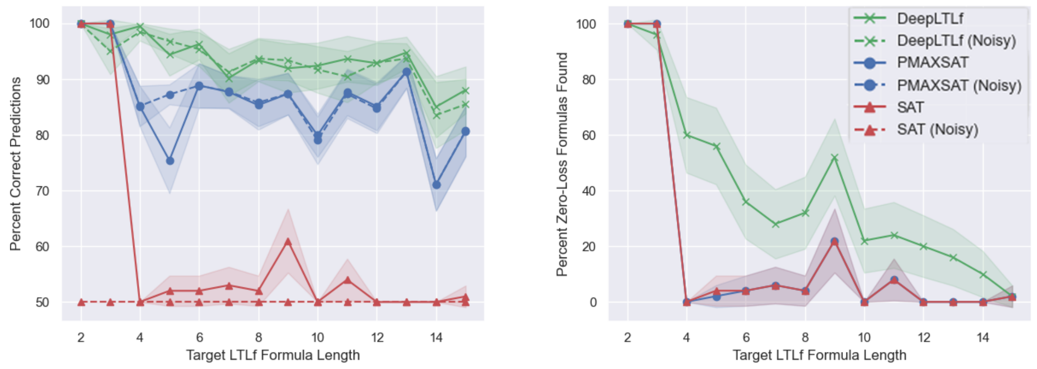

Figure 2. Comparison of the performance of DeepLTL$_f$, and SAT and PMAX-SAT-based methods on the test set after training on data with and without noise. An accuracy of 50% for the SAT method indicates the run timed out. 95% confidence intervals shown.

characteristic if the set uniquely defines a minimal DFA over a fixed number of states, N. Including a characteristic sample as part of the training data encouraged each method to produce a formula of greater complexity. We mixed the characteristic sample with uniformly sampled random traces such that $|\Pi_P| = |\Pi_N| = 500$ for all formulas. We explored the alternative of using solely random traces, but found that the algorithms reliably found shortcut solutions that did not capture the true target formula. The mix of the characteristic sample and random traces produced much more reliable results. Lastly, the labels of 1% of the total 1000 traces were swapped to produce a noisy dataset. We resampled the random traces for each formula to create the test data.

The last timestep of each trace in the characteristic sample was repeated such that all traces had length 15. Repeating the last timestep of a trace is guaranteed not to change its truth values with respect to a qualitative formula, as qualitative formulas define stutter-invariant languages (Peled & Wilke, 1997). However, padding may change the truth values of traces for metric formulas. Because of the complexities of training on variable length data, we chose to only use qualitative formulas in our experiments. Accordingly, both DeepLTL$_f$ and the SAT-based methods were modified to only produce qualitative formulas.

Since our intention was to test the scaling capabilities of each method, these datasets were produced with larger formulas (max size 15 vs 11) than those tested by Camacho & McIlraith (2019).

Procedure Each method was given a maximum runtime of 5 minutes per formula. DeepLTL$_f$ was allowed 3 network architectures each with 1 random restart. Table 3 displays the chosen architectures. The batch size was set at 100 and the learning rate at 0.005. Each network was run for 3000 epochs or until accuracy after discretization reached 100%. The sigmoid and ReLU activations were

Table 3. The 3 DeepLTL$_f$ network architectures used in the experiment on random formulas. The filter assignments denote the number of filters in each layer with the input layer on the left and the output layer on the right.

Network	Layers	Filter Assignment
1	1	1
2	2	$3 \rightarrow 1$
3	3	$5 \rightarrow 5 \rightarrow 1$

linearly annealed with rates $\beta_d = 0.01$ and $\alpha_d = -7\text{e}{-5}$ respectively. All hyperparameters for DeepLTL$_f$, including network architectures, were chosen via experimentation on held out random formulas. Since the data was generated solely from qualitative formulas, metrics weights (W_M) were removed from DeepLTL$_f$. The SAT-based methods were run with solvers from Z3 (De Moura & Bjørner, 2008). If a SAT-based method failed to produce any formula in the alloted time, we defaulted to the formula $true$ (which gives 50% accuracy). Experiments were conducted on Debian machines with Intel Core i5-4690 CPUs at 3.5 GHz and 8 GB of RAM.

4.2. Results

We used accuracy, defined as the percentage of correctly classified traces, as a performance metric to compare approaches. Figure 2 shows the performance of DeepLTL$_f$, SAT, and PMAX-SAT on the test datasets after training on the original and noisy datasets. In both settings, DeepLTL$_f$ consistently produced formulas with high accuracy over all target formula lengths. The standard SAT approach by Camacho & McIlraith (2019) began to time out on most formulas past a target formula length of 3. While Camacho & McIlraith (2019) test the scalability of their approach using an active learning setup with no more than 40 traces per formula, we used passive learning and 1000 traces per formula which caused the method to time out on much smaller formulas. Additionally, the SAT approach timed out

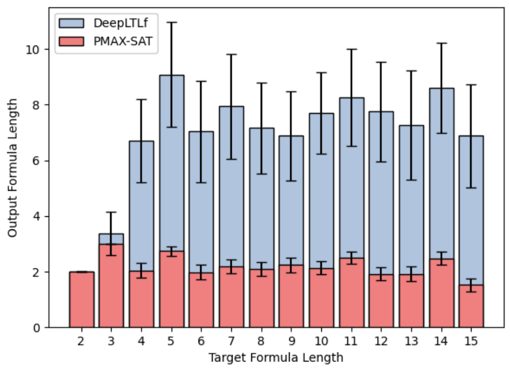

Figure 3. The length of formulas produced by DeepLTL$_f$ and PMAX-SAT on the non-noisy data with 95% confidence intervals shown.

on all formulas in the noisy setting. However, our PMAX-SAT variant performed significantly better than the standard SAT approach. In both settings and over all target formula lengths, our PMAX-SAT variant produced formulas with only slightly worse accuracy than DeepLTL$_f$.

Further investigation into the formulas produced by DeepLTL$_f$ and PMAX-SAT revealed that DeepLTL$_f$ produced larger formulas on average (Figure 3). PMAX-SAT was unable to produce formulas larger than size 3 in the allotted time. Unlike the SAT-based methods, DeepLTL$_f$ is not guaranteed to produce an optimally compact formula. With larger formulas, DeepLTL$_f$ is able to fit to more patterns in the data and achieve higher accuracy than the SAT-based methods. However, in some instances, DeepLTL$_f$ produced very large, unintelligible formulas. For instance on data for a target formula $a \vee \mathsf{G}\,\neg c$, one DeepLTL$_f$ network learned the formula $(a \vee (((a \wedge \neg c) \vee (\neg b \wedge \neg c)) \wedge \mathsf{G}\,\neg c))\,\mathsf{R}\,(a \vee ((a \vee (b \wedge \neg c)) \wedge \mathsf{G}\,\neg c) \vee (((a \wedge \neg c) \vee (\neg b \wedge \neg c)) \wedge \mathsf{G}\,\neg c))$. This formula perfectly classified the data, but clearly its size is undesirable. We set a maximum formula-size threshold of 25 as an informal notion of readability. When selecting an output formula from those produced by the 3 networks we trained for each target formula (Table 3), we ignored those larger than 25.

We also calculated the percentage of output formulas with zero loss for DeepLTL$_f$ and PMAX-SAT (the standard SAT approach only produces formulas with zero loss). The percentage of zero-loss formulas for PMAX-SAT closely tracked the percentage of formulas found by SAT, indicating that PMAX-SAT was able to find a perfect formula in nearly all cases for which SAT found a formula. Notably, DeepLTL$_f$ was able to find significantly more zero-loss formulas at greater target formula lengths than the others.

As an example of the qualities of the formulas produced by the different methods, consider the target formula $b \vee \mathsf{G}\,\neg a \vee (b\,\mathsf{R}\,a)$ of size 8. When given the data for this formula, DeepLTL$_f$ produced the exact target formula. The SAT method timed out and the PMAX-SAT method produced the formula b, which gave 91% accuracy. While b captures part of the target formula and classifies a majority of the traces correctly, much of the original formula's complexity is lost. Since the SAT-based methods could only produce formulas up to size 3 in the allotted time, the formulas produced by these methods often lacked relevant components. DeepLTL$_f$'s ability to produce larger formulas in a shorter amount of time enabled it to find more complex formulas that better fit the data.

5. Discussion

We presented DeepLTL$_f$, a neural network solution to the LTL$_f$ learning problem, and evaluated its ability to scale to larger formulas as well as its robustness to noise. When tested on data sampled from random formulas, we found that DeepLTL$_f$ is capable of producing more accurate formulas on more complex tasks than the SAT-based approaches. When tested on a noisy version of the same data, we found that DeepLTL$_f$'s performance was minimally affected.

However, there are a number of points at which DeepLTL$_f$ may fail to produce both a highly accurate and interpretable formula. During formula extraction, information can be lost when the activations of the network are discretized. The extracted formulas were on average 1% less accurate than the trained DeepLTL$_f$ networks. We sought to increase the probability the networks would learn representations that discretize well by annealing the activation functions and using random restarts.

Additionally, DeepLTL$_f$ filters are highly expressive. A network architecture consisting of a small set of filters can represent a multitude of LTL$_f$ formulas. Larger network architectures tended to learn formulas that were too large for human readability. Comparing multiple network architectures on any given dataset helped to alleviate this issue.

Lastly, constructing and minimizing the temporal truth table in the formula-extraction step can require significant computational effort when the number of filters or propositions is large. The number of rows in the truth table is exponential in these values. Nevertheless, our experiments indicate that DeepLTL$_f$ does not suffer from scaling issues to the same degree as existing approaches.

Restricting the expressiveness of DeepLTL$_f$ filters would help to address these issues. A smaller space of expressions would limit information loss during discretization, reduce the probability of unintelligible formulas, and allow for a more efficient formula-extraction procedure. We leave these topics for further research.

References

Camacho, A. and McIlraith, S. A. Learning interpretable models expressed in linear temporal logic. In *Proceedings of the International Conference on Automated Planning and Scheduling*, volume 29, pp. 621–630, 2019.

Cha, B., Iwama, K., Kambayashi, Y., and Miyazaki, S. Local search algorithms for partial maxsat. *AAAI/IAAI*, 263268, 1997.

De Giacomo, G. and Vardi, M. Y. Linear temporal logic and linear dynamic logic on finite traces. In *IJCAI'13 Proceedings of the Twenty-Third International Joint Conference on Artificial Intelligence*, pp. 854–860. Association for Computing Machinery, 2013.

De Moura, L. and Bjørner, N. Z3: An efficient smt solver. In *International conference on Tools and Algorithms for the Construction and Analysis of Systems*, pp. 337–340. Springer, 2008.

Duret-Lutz, A., Lewkowicz, A., Fauchille, A., Michaud, T., Renault, E., and Xu, L. Spot 2.0—a framework for LTL and ω-automata manipulation. In *International Symposium on Automated Technology for Verification and Analysis*, pp. 122–129. Springer, 2016.

Fukushima, K. Neural network model for a mechanism of pattern recognition unaffected by shift in position-neocognitron. *IEICE Technical Report, A*, 62(10):658–665, 1979.

Kasenberg, D. and Scheutz, M. Interpretable apprenticeship learning with temporal logic specifications. In *2017 IEEE 56th Annual Conference on Decision and Control (CDC)*, pp. 4914–4921. IEEE, 2017.

Kim, J., Muise, C., Shah, A., Agarwal, S., and Shah, J. Bayesian inference of linear temporal logic specifications for contrastive explanations. In *IJCAI*, pp. 5591–5598, 2019.

Kingma, D. P. and Ba, J. Adam: A Method for Stochastic Optimization. In *ICLR 2015*, 2015.

Li, X., Vasile, C.-I., and Belta, C. Reinforcement learning with temporal logic rewards. In *2017 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pp. 3834–3839. IEEE, 2017.

Littman, M. L., Topcu, U., Fu, J., Isbell, C., Wen, M., and MacGlashan, J. Environment-independent task specifications via GLTL. *arXiv preprint arXiv:1704.04341*, 2017.

Neider, D. and Gavran, I. Learning linear temporal properties. In *2018 Formal Methods in Computer Aided Design (FMCAD)*, pp. 1–10. IEEE, 2018.

Parekh, R. and Honavar, V. Learning dfa from simple examples. *Machine Learning*, 44(1-2):9–35, 2001.

Peled, D. and Wilke, T. Stutter-invariant temporal properties are expressible without the next-time operator. *Information Processing Letters*, 63(5):243–246, 1997.

Pnueli, A. The temporal logic of programs. In *18th Annual Symposium on Foundations of Computer Science (sfcs 1977)*, pp. 46–57. IEEE, 1977.

Pnueli, A. and Rosner, R. On the synthesis of a reactive module. In *Proceedings of the 16th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pp. 179–190, 1989.

Rautenberg, W. *A concise introduction to mathematical logic*. Springer, 2010.

Rudell, R. L. Multiple-valued logic minimization for PLA synthesis. Technical report, California University Berkeley Electronics Research Lab, 1986.

Vazquez-Chanlatte, M., Jha, S., Tiwari, A., Ho, M. K., and Seshia, S. Learning task specifications from demonstrations. In *Advances in Neural Information Processing Systems*, pp. 5367–5377, 2018.