Learning Finite Linear Temporal Logic Formulas

Homer Walke

Advisor: Michael Littman
Reader: Daniel Ritchie

Department of Computer Science
Brown University
Providence, RI
May 2021

Contents

Abstract

We present two methods for synthesizing finite linear temporal logic (LTL_f) specifications from labeled traces of system behavior. The first method reduces the problem to a partial maximum satisfiability problem (PMAX-SAT). The second method, NeuralLTL_f, introduces a novel recurrent neural operator designed to imitate the function of LTL_f operators. We train networks composed of the neural operator to classify the traces. Then, we extract an LTL_f formula from the learned weights by discretizing the activations of the network. We evaluate our methods on synthetic data, comparing their scalability with respect to formula size as well as their robustness to noisy data.

Disclaimer: This thesis is based on collaborative work with Daniel Ritter, Carl Trimbach, and Michael Littman [27].

1 Introduction

Demonstrations are a fundamental tool for teaching humans skills. However, leveraging demonstrations for teaching artificial agents presents a number of challenges. One challenge lies in choosing the agent's internal representation for the demonstrated behavior. A number of avenues have been explored, including learning a policy directly by behavior cloning or learning a reward function by inverse reinforcement learning [2]. However, both policies and reward functions can be uninterpretable. They may employ a significant number of parameters, which makes it difficult to easily understand the behavior they produce. Finite linear temporal logic (LTL_f) provides an alternative, more interpretable representation for sequential behavior since LTL_f formulas closely map to natural language [8]. Moreover, LTL_f formulas are symbolic representations that allow for high-level reasoning and manipulation. In this thesis, we examine the problem of learning an LTL_f formula from labeled traces of behavior.

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.*

There are a number of motivations for the LTL_f learning problem. LTL_f learning is useful in general classification tasks on sequential data. While recurrent neural networks are highly effective at processing time series data, interpreting the functions they learn is typically impossible. LTL_f learning is useful in sequential classification problems where one requires that the learned classifier is human-understandable or suitable for efficient reasoning and manipulation.

In formal methods, LTL_f learning is central to the task of specification mining or extracting temporal logic formulas from the execution traces of programs [16]. These specifications can be used to formally verify the correctness of the programs. In a related formal methods application, LTL_f learning can provide formulas for reactive synthesis, or the automated construction of finite-state machines where all executions satisfy a temporal logic specification [5].

Finally, as discussed initially, LTL_f learning is useful for teaching an artificial agent from demonstrations. Prior work on learning from demonstration has focused on learning policies, reward functions, or dynamics models [2]. However, such representations are not as interpretable as LTL_f formulas nor as suitable for symbolic manipulation. Additionally, these representations depend on the states of a specific environment, so transferring them to new environments is non-trivial. LTL_f on the other hand, is environment-independent [18]. With an LTL_f formula, one can derive rewards for an environment that maximize the probability of satisfying the formula [3, 13, 17].

Several prior approaches have reduced the LTL_f learning problem to a SAT problem [4, 20]. With highly-optimized SAT solvers, these methods can quickly produce small formulas when given a small number of traces. However, since the size of the SAT problem scales exponentially with the number of traces and allowed length of the LTL_f formula, these methods can become computationally intractable. Additionally, Camacho and McIlraith [4]'s SAT method fails when given traces that are not perfectly separable with a formula of the specified size, as if often the case when the data are noisy.

We describe a modification to Camacho and McIlraith's method where we encode the LTL_f learning problem as a partial maximum satisfiability problem (PMAX-SAT). This modification allows the method to find optimally accurate formulas even when no formula of the specified size perfectly classifies the data. Then, we introduce a new method for LTL_f learning, called NeuralLTL_f. In the first phase of NeuralLTL_f, we train a neural network consisting of a specialized recurrent operator on the traces. After training the network, we discretize its activations and extract an LTL_f formula that we simplify using logic-minimization techniques. We test the PMAX-SAT method and NeuralLTL_f on synthetic data. We find the PMAX-SAT method and NeuralLTL_f can produce formulas on noisy data where the SAT method times out, and NeuralLTL_f scales to produce larger formulas than either of the SAT-based methods.

1.1 Related Work

Camacho and McIlraith [4] reduce the LTL_f learning problem to a SAT problem and their method is the basis of our PMAX-SAT approach. Their SAT encoding is described in more detail in Section 3.1. Neider and Gavran [20] similarly reduce the problem to SAT, however they also show a technique for combining their SAT encoding with decision trees that scales effectively to larger trace sets. Kim et al. [14] define a Bayesian probabilistic model where a maximum a posteriori estimate corresponds to a solution to the LTL_f learning problem. However, they use LTL_f templates to reduce the space of possible formulas. Our methods produce formulas in the full space of LTL_f.

Also related to NeuralLTL_f is the well-studied problem of extracting a finite-state machine from a learned RNN. Recently, Michalenko et al. [19] showed a strong correspondence between the internal representations of RNNs trained to recognize a regular language and the state of the minimal DFA for the language. The correspondence justifies a number of attempts to decode the hidden state activations of an RNN into the states of a DFA, largely via clustering algorithms [7, 21], although Weiss et al. [28] achieved high accuracy by treating the RNN as an oracle for active learning. NeuralLTL_f is similar to these approaches in its goal of extracting a formal language representation from an RNN, but its formula extraction procedure is specialized for LTL_f.

2 Linear Temporal Logic

Linear temporal logic (LTL) is a type of logic that allows for reasoning about the behavior of a system through time [23]. LTL augments propositional logic with temporal operators that specify *when* a proposition holds. Accordingly, LTL formulas are evaluated over sequential data or traces. A trace is a sequence of truth assignments to a set of propositions. A trace may satisfy or violate an LTL formula. LTL is defined on infinite-length traces, however some systems are more accurately modeled with finite-length traces. LTL_f is a variant of LTL defined on finite-length traces [8], and LTL_f is the focus of this thesis.

An LTL_f formula can be evaluated at every timestep t of a trace where $0 \leq t < T$ and T is the trace length. $\pi, t \models \phi$ denotes that the formula ϕ holds at timestep t in trace π. However, a trace *satisfies* a formula only when the formula holds at the initial timestep. That is, π satisfies ϕ when $\pi, 0 \models \phi$.

All temporal operators in LTL_f are derived from the two fundamental temporal operators next (X) and until (U). The fragment of LTL_f consisting of only next and next-derived operators is called *metric* LTL_f, and the fragment consisting of only until and until-derived operators is called *qualitative* LTL_f [12]. $\mathsf{X}\,\phi$ denotes that ϕ will hold in the **next** timestep, while $\phi \,\mathsf{U}\, \psi$ denotes that ϕ must hold **until** ψ becomes true:

$$\begin{aligned} \pi, t \models \mathsf{X}\,\phi &\iff \pi, t+1 \models \phi \text{ and } t+1 < T \\ \pi, t \models \phi \,\mathsf{U}\, \psi &\iff \pi, k \models \psi \text{ for some } t \leq k < T \\ &\qquad \text{and } \pi, j \models \phi \text{ for all } t \leq j < k. \end{aligned}$$

The operators eventually (F), globally (G), release (R), and weak until (W) are all derived from until.

$$\begin{aligned} \mathsf{F}\,\phi &\iff \textit{true}\,\mathsf{U}\,\phi \\ \mathsf{G}\,\phi &\iff \neg\,\mathsf{F}\,\neg\phi \\ \phi\,\mathsf{R}\,\psi &\iff \neg(\neg\phi\,\mathsf{U}\,\neg\psi) \\ \phi\,\mathsf{W}\,\psi &\iff (\phi\,\mathsf{U}\,\psi) \vee \mathsf{G}\,\phi \end{aligned}$$

The operator weak next (N), unique to LTL_f, is derived from next.

$$\mathsf{N}\,\phi \iff \mathsf{X}\,\phi \vee \neg\,\mathsf{X}\,\textit{true}$$

In LTL_f, $\neg\,\mathsf{X}\,\textit{true}$ holds at only the last timestep, so $\mathsf{N}\,\phi$ denotes that ϕ must hold at the next time step *or* the next time step does not exist. Table 1 gives a summary of each operator's semantics.

Finally, we define the length of an LTL_f formula.

Definition 2 (Length of an LTL_f formula)**.** *The length of an LTL_f formula is the sum of the number of temporal operators, binary logical operators, and propositions in the formula.*

Table 1: LTL_f operators and their natural language equivalents.

LTL_f Operator	Symbol	Natural Language
Until	$\phi \,\mathsf{U}\, \psi$	ϕ holds up to, but not necessarily including, the point where ψ becomes true.
Weak until	$\phi \,\mathsf{W}\, \psi$	ϕ holds up to, but not necessarily including, the point where ψ becomes true, or ϕ holds forever.
Next	$\mathsf{X}\, \phi$	ϕ holds in the following timestep.
Weak next	$\mathsf{N}\, \phi$	ϕ holds in the following timestep, or the current timestep is the last timestep.
Eventually	$\mathsf{F}\, \phi$	ϕ holds at some point in the future.
Globally	$\mathsf{G}\, \phi$	ϕ holds at the current and all subsequent timesteps.
Release	$\phi \,\mathsf{R}\, \psi$	ψ holds up to and including the point where ϕ becomes true, or ψ holds forever.

3 Methods

3.1 Partial MAX-SAT

Camacho and McIlraith [4] reduce the LTL_f learning problem into a Boolean satisfiability (SAT) problem. We investigated a modification to the approach where we instead reduced LTL_f learning to partial maximum satisfiability (PMAX-SAT).

3.1.1 Reduction into SAT

The goal in a SAT problem is to find a setting of the variables in a Boolean formula such that the formula evaluates to *true*. Given a set of labeled traces, Camacho and McIlraith produce a Boolean formula, h, where a satisfying assignment of the variables corresponds to an LTL_f formula that distinguishes the traces. Then, they use a SAT solver to find a satisfying assignment to h from which they recover the LTL_f formula, ϕ.

Their procedure for encoding the LTL_f learning problem into SAT is based on the correspondence between LTL_f formulas and automata. Specifically, any LTL_f formula can be converted into an alternating finite automaton (AFA) that accepts and rejects the same traces as the formula [26]. However, the converse does not hold. Not every AFA can be converted into an equivalent LTL_f formula. Camacho and McIlraith define a set of AFA *skeletons* that correspond to LTL_f operators. A skeleton is a fragment of an AFA that performs the same operation as an LTL_f operator. AFA skeletons can be fit together to compose a full AFA and, importantly, AFA constructed from skeletons can always be converted into equivalent LTL_f formulas.

Camacho and McIlraith's SAT encoding procedure models the process of constructing an AFA from AFA skeletons. Given a maximum formula size N, the procedure first defines a series of Boolean variables that determine which N skeletons will be used and how they fit together. Then, the procedure constructs a Boolean formula, h, consisting of a series of clauses over these variables. One set of clauses enforces that the AFA is well-constructed. Another set of clauses restricts the size of the AFA, and thus the size of the corresponding LTL_f formula. Finally, another set of clauses ensures the AFA accepts the positive traces and rejects the negative traces. h is given to a SAT solver which finds a satisfying assignment to the variables. The skeletons used in the construction of the AFA are determined from this variable setting and the corresponding LTL_f formula is recovered.

3.1.2 Learning Algorithms

Camacho and McIlraith examine the applications of their method to both passive and active learning. Passive learning models the traditional classification setting where the task is to find a hypothesis that distinguishes a set of positive and negative examples. They define an algorithm for passive learning that ensures their method produces the minimum size formula that classifies the traces. The algorithm incrementally increases the formula size, N, given to the SAT encoding procedure until the SAT solver finds a satisfying assignment (Algorithm 1).

Algorithm 1 Passive Learning

```
Input: trace sets $\Pi_P$ and $\Pi_N$
$N \leftarrow 1$
$h \leftarrow$ SAT encoding of $\Pi_P$ and $\Pi_N$ for formula size $N$
while $h$ is unsatisfiable do
    $h \leftarrow$ SAT encoding of $\Pi_P$ and $\Pi_N$ for formula size $N$
    $N \leftarrow N + 1$
end while
return $\mathsf{LTL}_f$ formula, $\phi$, recovered from satisfying assignment for $h$
```

They also define an active learning algorithm. Active learning assumes access to an oracle that answers *membership queries* and *equivalence queries*. Membership queries simply ask whether a trace satisfies the target formula. Equivalence queries ask whether a formula is equivalent to the target formula. If the formula is not equivalent, the oracle provides a counterexample: a trace on which the guessed formula and target formula disagree. The active learning algorithm uses the passive learning algorithm as a subroutine. The algorithm begins by performing passive learning on empty positive and negative trace sets, Π_P and Π_N. The resulting output formula is used in an equivalence query to the oracle, which gives a counterexample if the output formula is not equivalent to the target formula. The counterexample is added to the trace sets and the process repeats until the target formula is found (Algorithm 2).

Algorithm 2 Active Learning

```
Input: Oracle for a target formula
$N \leftarrow 1$
$\Pi_P \leftarrow \emptyset$
$\Pi_N \leftarrow \emptyset$
$\phi \leftarrow$ formula output from passive learning on $\Pi_P, \Pi_N$
while $\phi$ is not equivalent to the target formula do
    $\pi \leftarrow$ counterexample provided by the oracle
    Add $\pi$ to $\Pi_P$ or $\Pi_N$
    $\phi \leftarrow$ formula output from passive learning on $\Pi_P, \Pi_N$
end while
return $\phi$
```

We used the passive learning setup for Camacho and McIlraith's method in our experiments (Section 4).

3.1.3 PMAX-SAT Modification

A limitation of Camacho and McIlraith's approach is that the SAT solver is unable to produce any formula if the trace sets are not distinguishable with a formula of the specified size, N. This limitation prevents the application of the method to noisy data. We would like a method that produces a formula of size N or below that achieves the the best accuracy possible on the trace sets, which may not be 100%. In terms of the SAT encoding, we want the solver to satisfy the maximum number of clauses that correspond to accepting the positive traces and rejecting the negative traces. However, we still need to ensure the solver satisfies the clauses that enforce the validity and size of the AFA. The partial maximum satisfiability problem (PMAX-SAT) describes the task of satisfying the maximum number of a set of "soft" clauses, while also satisfying a set of "hard" clauses [6]. We implemented a modification of Camacho and McIlraith's approach based on PMAX-SAT. We labeled all the clauses that enforce the acceptance and rejection of positive and negative traces as soft and the rest of the clauses as hard. We then switched the SAT solver with a PMAX-SAT solver. In this way, the satisfying assignment produced by the solver corresponds to the formula that achieves the optimal accuracy on the trace sets. We used the passive learning algorithm for this PMAX-SAT variant in our experiments. Section 4 shows the results.

3.2 NeuralLTL_f

The semantics of LTL_f presented in Section 2 suggest a connection to recurrent neural networks. Particularly, the evaluation of an until or until-derived operators recursively depends on their evaluation at future timesteps. Based on this connection, we define a recurrent neural operator that imitates the application of temporal operators in LTL_f. We take the simple RNN operator and remove the hidden layer so that the recurrent input directly reads from the output at the previous timestep, simulating the operation of an until operator. We also apply the operator to two timesteps at a time, analogous to a 1D convolution with a filter width of two. This modification simulates the manner in which next operators specify behavior one timestep into the future. Finally, we apply the recurrent operator backwards along the trace to model how temporal operators depend on their evaluation at *future* timesteps. Equation 1 provides a formal definition of the operation.

Using the terminology of convolutional neural networks, we call the weights of our operator a filter [11]. We can compose multiple filters into the layers of a network. In this way, filters are applied to the results of previous filters, similar to the nesting of operators in an LTL_f formula. The input to the network is a trace where the truth values of the trace are interpreted as 0 and 1 for *true* and *false*. The output of the network is a sequence

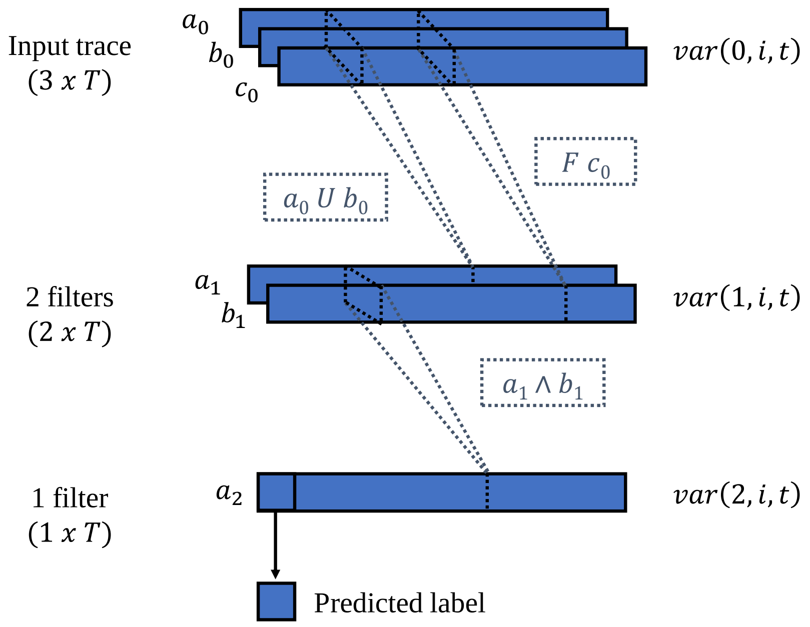

Figure 1: A NeuralLTL$_f$ network that encodes the formula $(a \,\mathsf{U}\, b) \wedge \mathsf{F}\, c$. Solid boxes are the input trace and output activations. Dashed lines represent the application of the filters. The formulas in the dashed boxes represent formula fragments learned by each filter.

representing the network's prediction of the truth value at every timestep of the trace. Just like an LTL$_f$ formula, we take the first timestep of the output sequence to be the predicted label of the trace. We compare the predicted label against the target label to compute a loss that we minimize via gradient descent. Figure 1 provides a visual overview of a NeuralLTL$_f$ network. After the network is trained, the filter weights represent a continuous version of an LTL$_f$ formula that classifies the traces. We discretize the activations of the trained network to extract an LTL$_f$ formula.

3.2.1 Network Weights

A filter is composed of sets of weights that allow for the expression of both qualitative and metric temporal operators. While layers in the network can consist of multiple filters, we restrict the last layer to a single filter so that the output consists of a single truth value per timestep. Filters act on sequences of values for a set of *variables*, indexed by j. These variables might be the propositions in the input trace, in the case of the first layer, or the activations of the filters in a previous layer. We use $\text{var}(l, i, t)$ to denote the activation of filter i at timestep t in layer l, and we define the input trace as $\text{var}(0, i, t)$. The weights of a filter are defined as follows.

- $W_P(l, i, j)$ 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.
- $\text{var}(l-1,j,T+1)$ and $\text{var}(l,i,T+1)$ are base case values.

We apply the weights of a filter to a sequence using the following formula.

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

The nonlinear activation function, σ, is the sigmoid. We also apply the function $\delta(x) = \max(x, \alpha x)$ to the W_Q weight, where α is a hyperparameter. δ biases W_Q towards positive values since we require that W_Q is positive for formula extraction (Section 3.2.2). Each filter is a linear classifier that predicts a truth value at a timestep based on the the values of the propositions at the current timestep, the values of the propositions at the next timestep, and the recursive output of the filter at the next timestep. The base case values for the recursion, $\text{var}(l-1,j,T+1)$ and $\text{var}(l,i,T+1)$, are parameters learned along with the weights.

Filters can represent all LTL_f operators as well as standard logical operators (Table 2). However, because the weights are continuous, a filter may also represent operations that do not cleanly map to a temporal operator. When training the network we employ several techniques to minimize this undesirable outcome (see Section 3.2.4).

3.2.2 Conversion from Network Weights to Formula

We now describe a procedure for extracting an LTL_f formula from a trained NeuralLTL$_f$ network. The procedure consists of two steps. First, we convert the weights of each filter into an intermediate representation we call a *temporal truth table*. Then we convert the temporal truth tables into an LTL_f formula.

A temporal truth table, f, represents a discrete approximation of the function encoded by a filter. The table restricts all input and output of the filter to $\{0,1\}$. The rows of the table record all the possible discrete inputs to the filter along with the corresponding discrete filter output. The table has columns for the values of each of the n propositions at the

Table 2: Example filter weights for LTL_f operators. We show one NeuralLTL$_f$ filter, i, applied to two truth value sequences representing the LTL_f formulas ϕ and ψ. 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.

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

current timestep, x_j, the values of propositions at the next time time step, m_j, and a column representing the output of the filter in the next time step, τ. We call x_j, m_j, and τ the propositional, metric, and temporal bits respectively. To create a discrete filter output we set the activation function, σ, to the binary step function, $\mathbb{1}_{[0,\infty)}$. We also set $\delta(x) = \max(x, 0)$ since we require that W_Q is positive to create a valid truth table (discussed later).

Then, filling in the temporal truth table for a filter involves applying each row of discrete input to the filter weights and recording the discrete output, multiplying x_j with W_P, m_j with W_M, and τ with W_Q. Additionally, separate from the table, we define additional bits Ω and ω_j, derived by applying the binary step function to the base case values $\text{var}(l,i,T+1)$ and $\text{var}(l-1,j,T+1)$. The result of the first step is a table, f, and the bits Ω and ω. Algorithm 3 describes the first step. Table 3 gives an example temporal truth table.

The second step converts f, Ω, and ω into an LTL_f formula. Specifically, we derive an LTL_f formula in a form we call *temporal normal form* (TNF). TNF describes a formula in the form $\phi\,\mathsf{U}\,\psi$ or $\phi\,\mathsf{W}\,\psi$. ϕ and ψ are formulas in full disjunctive normal form [24] over the set of all propositions and all propositions prepended by a next (X) or weak next operator (N). That is, ϕ and ψ are a disjunction of conjunctions where each proposition, along with each proposition prepended by a next or weak next operator, appears exactly once.

First, we use Ω to determine whether the formula will be in the form $\phi\,\mathsf{U}\,\psi$ or $\phi\,\mathsf{W}\,\psi$. We use U if $\Omega = 0$ and W if $\Omega = 1$. Ω records the filter's behavior at the end of the trace, which determines whether an until operator is weak or strong. Then, we construct ϕ by taking the disjunction of the conjunction of all the propositional and metric bits in rows where the output is 1 and $\tau = 1$. ϕ represents all the inputs that result in *true* when the output of the filter in the next timestep is *true*. We construct ψ by taking the disjunction of the conjunction of all the propositional and metric bits in rows where the output is 1 and $\tau = 0$.

Algorithm 3 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
$\Omega \leftarrow \sigma(\text{var}(l, i, T+1))$
**for** $j \in \{1 \ldots n\}$ **do**
    $\omega_j \leftarrow \sigma(\text{var}(l-1, j, T+1))$
**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** $f, \Omega, \omega$
```

Table 3: A temporal truth table for the formula $x_1 \mathsf{U} x_2$. This table is an example of an ouput of Algorithm 3 and the input to Algorithm 4. Rows where the output is 0 are omitted for brevity.

$\Omega = 0$		$\omega_1 = 0$		$\omega_2 = 0$	
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

ψ represents all the inputs that result in *true* no matter the future value of the filter. Metric bits, m_j, in ϕ and ψ take the form $\mathsf{X}\, x_j$ if $\omega_j = 0$ and $\mathsf{N}\, x_j$ if $\omega_j = 1$, since whether a next operator is weak or strong is determined by behavior at the end of the trace. Rows where the output is 0 have no effect on the LTL_f formula. Algorithm 4 describes the second step.

Example of Conversion Procedure As an example, we will carry out the conversion procedure for a filter that represents the LTL_f formula $x_1 \,\mathsf{U}\, x_2$. We assume the filter has the weights listed in the first row of Table 2. First, we create a temporal truth table, f, by evaluating the filter for every setting of the propositional, metric, and temporal bits $(x_j, m_j$ and $\tau)$. We evaluate the filter using a discrete version of Equation 1, where σ is the binary step function and $\delta(x) = \max(0, x)$:

$$f[k] = \sigma(\Sigma_j W_P(l,i,j)x_j + \Sigma_j W_M(l,i,j)m_j + \delta(W_Q(l,i))\tau + b(l,i)).$$

Consider the bit setting $x_1 = 1, x_2 = 0, m_1 = 0, m_2 = 0, \tau = 1$. Plugging in this bit setting along with the filter weights we have:

$$f[k] = \sigma((1 \cdot 1 + 0 \cdot 2) + (0 \cdot 0 + 0 \cdot 0) + \delta(1) \cdot 1 - 1.5) = \sigma(0.5) = 1.$$

Thus, the output column for this bit setting in the temporal truth table is set to 1. We also apply the binary step function to the learned base case values of the filter, $\text{var}(l, i, T+1)$ and $\text{var}(l-1, j, T+1)$, to produce Ω and ω_j respectively. For the sake of the example, we will assume $\text{var}(l, i, T+1)$ and $\text{var}(l-1, j, T+1)$ are 0, even though the first row of Table 2 states they can take any value for this filter.

$$\Omega = \sigma(\text{var}(l, i, T+1)) = \sigma(0) = 0$$

$$\omega_1 = \sigma(\text{var}(l-1, 1, T+1)) = \sigma(0) = 0 \quad \omega_2 = \sigma(\text{var}(l-1, 2, T+1)) = \sigma(0) = 0$$

Table 3 shows the completely filled in table. We then convert the temporal truth table into an LTL_f formula. Since $\Omega = 0$, we know the formula will use an until operator and have the form $\phi \,\mathsf{U}\, \psi$. Now, we need to determine the sub-formulas ϕ and ψ. Each row in the temporal truth table with output 1 contributes a clause to ϕ or ψ. Let's once again consider the row corresponding to the bit setting: $x_1 = 1, x_2 = 0, m_1 = 0, m_2 = 0, \tau = 1$. This row represents the clause:

$$x_1 \wedge \neg x_2 \wedge (\mathsf{X} \neg x_1) \wedge (\mathsf{X} \neg x_2).$$

We use the next (X) operator to represent the metric bits, rather than the weak next (N) operator, because $\omega_1 = 0$ and $\omega_2 = 0$. We add this clause to ϕ, rather than ψ, since $\tau = 1$. So, we have:

$$\phi = \ldots \vee (x_1 \wedge \neg x_2 \wedge (\mathsf{X} \neg x_1) \wedge (\mathsf{X} \neg x_2)) \vee \ldots.$$

Repeating the process for every row with output 1 in the table results in the complete sub-formulas ϕ and ψ. We now have a TNF formula, $\phi \,\mathsf{U}\, \psi$, that simplifies to $x_1 \,\mathsf{U}\, x_2$.

Algorithm 4 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
```

3.2.3 Correctness of Conversion Procedure

First, we will show that every temporal truth table output by the conversion procedure can be converted into some LTL_f formula. The conversion procedure does not create *invalid* temporal truth tables or tables that encode logically impossible LTL_f formulas.

Theorem 1. *The conversion of a Neural*LTL_f *filter into a temporal truth table always results in a valid table.*

Proof. A temporal truth table encodes a logically impossible formula if it contains a setting of the x_j and m_j bits where the output is 1 when $\tau = 0$ and the output is 0 when $\tau = 1$. This table encodes an LTL_f expression where a setting of the propositions satisfies the expression (output is 1), no matter the expression's value in the next timestep ($\tau = 0$). But the same setting of the propositions violates the expression (output is 0), if the expression holds in the next timestep ($\tau = 1$). In other words, this table encodes a TNF formula (ϕ $\{\mathsf{U}, \mathsf{W}\}$ ψ) where a clause appears in ψ but explicitly does not appear in ϕ. But, according to the semantics of LTL_f, any clause that appears in ψ is implicitly in ϕ.

Now, we show that such a table cannot result from the conversion procedure. Consider the equation used to compute the output column, f, of the temporal truth table given a setting, k, of the x_j, m_j and τ bits (Algorithm 3).

$$f[k] = \sigma(\Sigma_j W_P(l,i,j)x_j + \Sigma_j W_M(l,i,j)m_j + \delta(W_Q(l,i))\tau + b(l,i))$$

Consider two settings of the bits, k_1 and k_2, where $f[k_1] = 0$ and $f[k_2] = 1$. k_1 and k_2 are identical except $\tau = 1$ in k_1 and $\tau = 0$ in k_2. Thus, k_1 and k_2 encode a logically impossible LTL_f expression. However, this situation implies:

$$f[k_1] < f[k_2].$$

Substituting the definitions of $f[k_1]$ and $f[k_2]$ and canceling identical terms gives:

$$\delta(W_Q(l,i)) \cdot 1 < \delta(W_Q(l,i)) \cdot 0$$

$$\delta(W_Q(l,i)) < 0.$$

But, since $\delta(x) = \max(0, x)$ during the conversion procedure, $\delta(W_Q(l,i))$ is never less than 0. Since the conversion procedure cannot create temporal truth tables with these settings, the procedure only creates valid tables. □

A correct conversion procedure is also LTL_f*-expression preserving*. That is, encoding an LTL_f formula in TNF as a temporal truth table and then converting the truth table into an LTL_f formula should result in a formula equivalent to the starting formula. By equivalence, we mean logical equivalence denoted by the operator $\equiv$ [24]. In the context of LTL_f, $g \equiv h$ if g and h have the same truth value on every trace.

Theorem 2. *The conversion procedure is* LTL_f*-expression preserving. Given any* LTL_f *expression* g *in TNF and its temporal truth table* f*, running the conversion procedure on* f *results in an* LTL_f *expression* h*, and* $g \equiv h$*.*

Proof. We can create a temporal truth table, f, for a TNF LTL_f formula g by evaluating g for every setting of the x_j, m_j, an τ bits in the table. Then, we can create a TNF LTL_f formula h by running the conversion procedure on f. Assume that g does not equal h. That is, there exists a trace that satisfies g and violates h or vice versa. It follows that g and h are syntactically different. If they were syntactically identical, they would accept and reject the same traces. However, since g and h are in TNF (ϕ $\{\mathsf{U}, \mathsf{W}\}$ ψ), they are structurally similar and their syntax can only differ in two ways. Either (1) g and h have syntactically different ϕ and ψ, or (2) g and h differ in the choice of U versus W.

In TNF, ϕ and ψ are in full disjunctive normal form. So if g and h have syntactically different ϕ and ψ, then one has a disjunct in ϕ or ψ where the other does not. Without loss of generality, assume g has a disjunct that h does not. This situation implies the row corresponding to the disjunct in f has output 1, since f was derived from g. But, according to the conversion procedure, h would also include the disjunct if the output of the row was 1, so there is a contradiction.

g and h may differ in the choice of U versus W. Without loss of generality, assume g has U and h has W. Then, $\Omega = 0$, since Ω is derived from g. However, according to the conversion procedure, if $\Omega = 0$, then h should have U, so there is a contradiction.

Thus, g and h cannot differ and the conversion procedure is LTL_f-expression preserving. □

Because the conversion procedure does not create invalid truth tables and is LTL_f-expression preserving, the procedure is correct.

3.2.4 Implementation Details

We implemented NeuralLTL_f in Tensorflow [1]. We used a binary cross-entropy loss optimized with Adam [15]. Additionally, we use several techniques that increase the accuracy and compactness of the formulas output by NeuralLTL_f.

Logic Minimization The TNF formulas output by the conversion procedure are typically too large for readability, so we employ a two-stage simplification procedure. We use the Espresso logic-minimization algorithm to initially simplify the formula using standard propositional logic simplification rules [25]. While not guaranteed to find the minimally sized formula, the Espresso algorithm uses heuristics to find close-to-optimal simplifications of formulas with large numbers of propositions more efficiently than exact methods. Then, we use the Spot LTL_f library to simply the formula according to LTL_f simplification rules [1]. In our experiments, the average percent reduction in formula size by Espresso was 91% and the average percent reduction by Spot was 51%.

Annealing and Random Restarts The conversion procedure discretizes the function represented by a trained NeuralLTL$_f$ network, leading to some information loss. However, we can train the network in a way that encourages learning a function that maintains high classification accuracy when discretized. Since the conversion procedure switches the sigmoid activation function, σ, to a binary step function, we linearly increase the steepness of the sigmoid after each epoch. Similarly, since the conversion procedure switches $\delta(x) = \max(x, \alpha x)$ with $\delta(x) = \max(x, 0)$, we linearly reduce α after each epoch. That is, we define σ and δ as

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

Given annealing rates α_d and β_d, we update the values of α and β at the end of each epoch by setting $\alpha = \alpha + \alpha_d$ and $\beta = \beta + \beta_d$. We also use random restarts, training the network multiple times with different random weight initializations. We select the trained network that has the highest accuracy after discretization.

Multiple Networks While NeuralLTL$_f$ does not depend on matching the structure of the network to the structure of the formula one expects to learn, a larger network has a greater risk of producing a formula that overfits the training data. Since the size of the minimal formula that describes the data is not known beforehand, we train several different network architectures on a given dataset to increase the probability of producing a formula close in size to the optimal formula. After extracting formulas from each network, we choose the smallest formula of the set of formulas with the highest accuracy on the test data.

[1]Spot is designed for LTL rather than LTL$_f$, but we show in the Appendix that we use Spot correctly for our experiments.

4 Experiments

Our experiments compared Camacho and McIlraith [4]'s SAT-based method, our PMAX-SAT modification, and NeuralLTL$_f$ on two criteria: (1) scalability with respect to LTL$_f$ formula size and (2) robustness to noisy data. Accordingly, we tested the methods on synthetic data that models LTL$_f$ formulas of varying sizes as well as a noisy version of the dataset.

4.1 Data

To create the synthetic data, we generated random qualitative LTL$_f$ formulas over 3 variables by a uniform sampling of the LTL$_f$ grammar. We generated 50 of each length ranging from 2 to 15 (see Definition 2 for how we calculate formula length), or as many as possible if the number of unique formulas for a given length was less than 50. Since our goal was to test the methods' ability to produce LTL$_f$ formulas, we rejected formulas that did not include a temporal operator, meaning there were no formulas of length 1.

Then, we generated traces that modeled each formula. We initially used simple rejection sampling, uniformly sampling traces and adding them to the positive or negative set depending on whether they satisfied the formula. However, we found that traces generated in this way did not fully represent the complexity of the target formula. All the methods were able to find short formulas with high classification accuracy, even on synthetic data for much longer formulas. Since our goal was to test the methods' ability to produce larger formulas, we turned to a new data generation technique where we mixed random rejection sampled traces with the *characteristic sample* (CS) for the formula's corresponding minimal deterministic finite-state automaton (DFA).

Definition 3. *A characteristic sample (CS) is the minimal set of labeled traces that uniquely defines a minimal DFA over a fixed number of states, N.*

By definition, the characteristic sample includes traces that are informative of each part of the target formula. So using the CS in the dataset prevents the methods from achieving perfect classification accuracy with a formula that has a simpler DFA representation than the target formula. To create the training data, we mixed the characteristic sample with rejection sampled random traces such that $|\Pi_P| = |\Pi_N| = 500$ for all formulas. To create the test data, we resampled the random traces for each formula. We swapped 1% of the labels to produce a noisy version of the training data.

We produced all the the random traces at length 15, and each trace in the characteristic samples was padded to length 15 by repeating the last timestep. Padding a trace is guaranteed not to change its truth values with respect to a qualitative formula, since qualitative formulas define stutter-invariant languages [22]. However, padding may change the truth

Table 4: The 3 NeuralLTL$_f$ network architectures used in the experiments. 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	$5 \rightarrow 5 \rightarrow 1$

values of traces with respect to metric formulas. Because of the difficulty in training an RNN on variable length data, we chose to only use qualitative formulas in our experiments. We modified both NeuralLTL$_f$ and the SAT-based methods to only produce qualitative formulas, removing the metric weights from NeuralLTL$_f$ and the weak and strong next operators from the SAT encoding.

4.2 Procedure

We limited each method to 5 minutes per target formula. For NeuralLTL$_f$, 5 minutes was sufficient to train 3 different network architectures (Table 4) for 3000 epochs with 1 random restart, though we halted training early if a network reached 100% discrete classification accuracy. The 3 network architectures were chosen to loosely cover the structure of the formulas we expected to learn, with 1 to 3 layers and 1 to 5 filters per layer. We used a batch size of 100, a learning rate of 0.005, and we set the annealing rates for σ and δ as $\beta_d = 0.01$ and $\alpha_d =$ -7e-5.

We implemented the SAT-based methods with the Z3 SAT solver [9]. If a SAT-based method timed out on a trace set, we defaulted to the formula *true* which has 50% accuracy on the synthetic trace sets. We conducted the experiments on Debian machines with Intel Core i5-4690 CPUs at 3.5 GHz and 8 GB of RAM.

4.3 Results

Figure 2 shows the test set classification accuracy of NeuralLTL$_f$, Camacho and McIlraith [4]'s SAT approach, and our PMAX-SAT variant after training on the original and noisy training data. NeuralLTL$_f$ produced formulas with slightly higher accuracy than PMAX-SAT over all target formula lengths. The SAT method began timing out on almost all trace sets past a target formula length of 3, since our experiments consisted of significantly more challenging learning problems than those tested by Camacho and McIlraith. Camacho and McIlraith test the scalability of their method using an active learning setup (see Section 3.1) on a maximum of 40 traces, while our experiments used a passive learning setup with

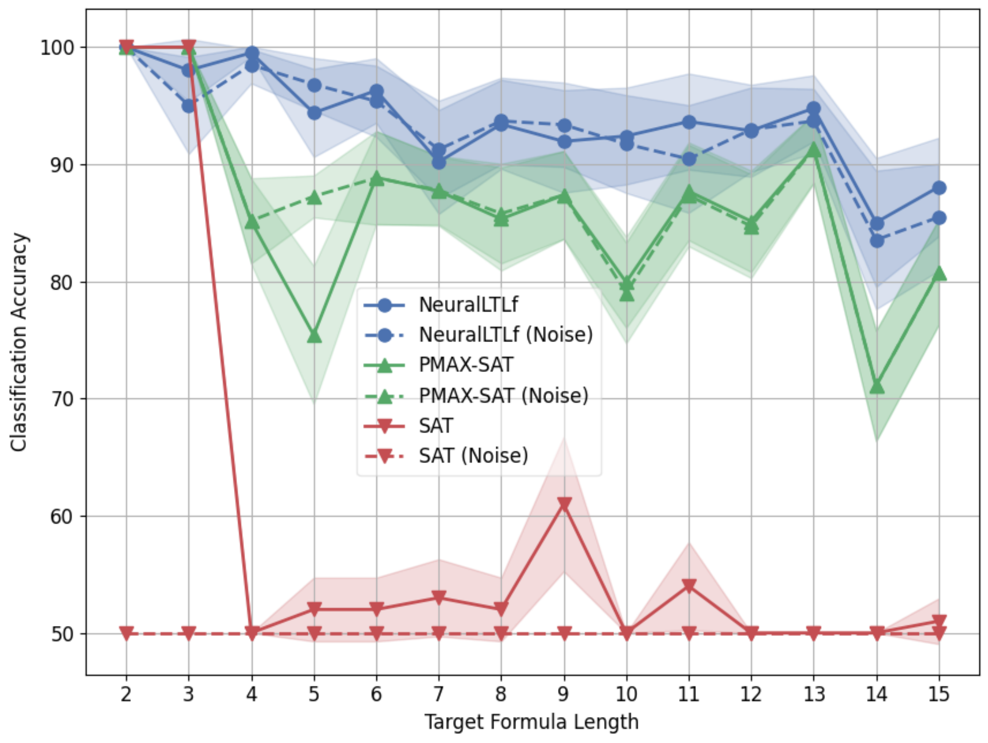

Figure 2: Test set classification accuracy. An accuracy of 50% for the SAT method indicates the run timed out. 95% confidence intervals shown.

1000 traces. We also attempted to learn formulas up to length 15 while they only attempt up to length 11.

Both NeuralLTL$_f$ and PMAX-SAT were largely unaffected by training on the noisy data. Both methods obtained approximately the same classification accuracy in the noisy setting as they did in the non-noisy setting. However, the SAT method failed to produce any formulas over all target formula lengths on the noisy data. This failure was expected because the SAT method is restricted to producing formulas with perfect accuracy on the training data. A minimum length formula that accommodates the 1% noise in the training data may be impractically large.

In Figure 3 we calculated the proportion of output formulas from each method that had zero-loss, or those that correctly classified every trace in the test set. This proportion was nearly identical for the SAT and PMAX-SAT methods, which indicates that in every case where the SAT solver could find a perfect formula, the PMAX-SAT solver could as well. NeuralLTL$_f$ found significantly more zero-loss formulas over all target formula lengths.

While the PMAX-SAT method produced formulas that had only slightly worse accuracy than NeuralLTL$_f$, inspection of the output formulas of both methods reveals that NeuralLTL$_f$ produced significantly longer formulas (Figure 4). In fact, PMAX-SAT was unable to produce a formula above length 3, even on the trace sets for target formulas of much greater length. The longer formulas produced by NeuralLTL$_f$ better fit the data and

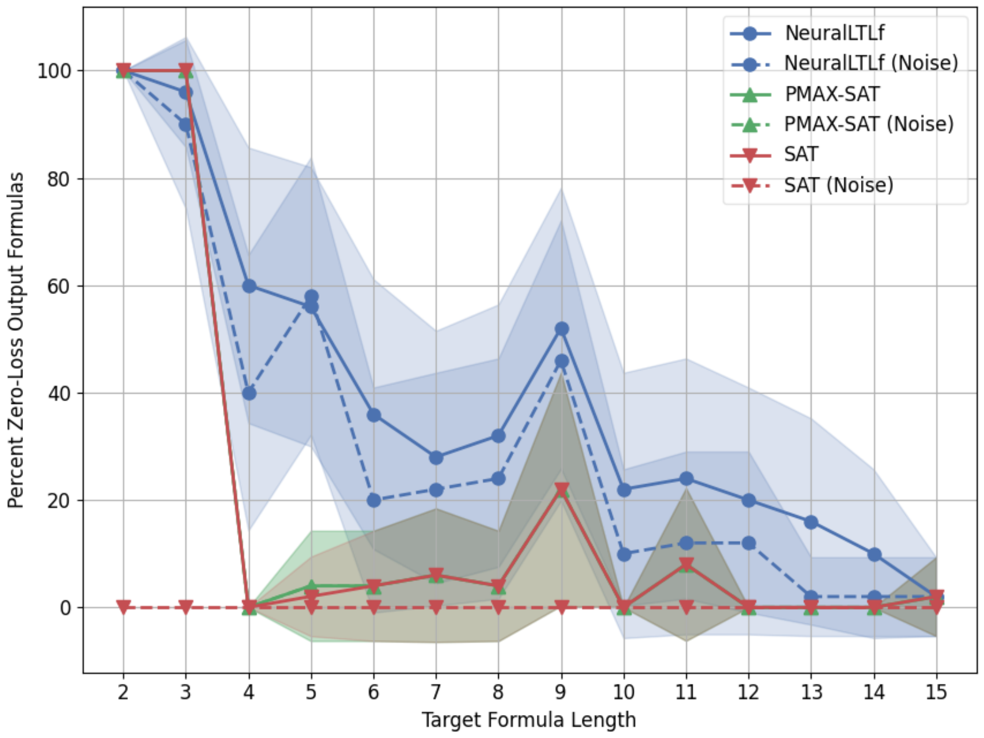

Figure 3: The percentage of output formulas that perfectly classified the test set. 95% confidence intervals shown.

achieved higher accuracy. For example, when given the trace sets for the target formula $b \vee \mathsf{G}\,\neg a \vee (b\,\mathsf{R}\,a)$, the SAT method timed out and the PMAX-SAT method produced the formula b, which gave 91% accuracy. However, NeuralLTL$_f$ produced the exact target formula, $b \vee \mathsf{G}\,\neg a \vee (b\,\mathsf{R}\,a)$. A short formula, like b, may achieve high accuracy on our synthetic data, but fails to capture the complexity of the target formula. NeuralLTL$_f$ found formulas that more closely matched the complexity of the target formulas.

However, in some cases NeuralLTL$_f$ produced very large formulas that clearly overfit the data. As an example, when given the trace set for the target formula $a \vee \mathsf{G}\,\neg c$, a NeuralLTL$_f$ network produced 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 is not human-readable. In order to discount formulas of this nature, we set a maximum formula-size threshold of 25. When selecting an output formula from those produced by the 3 networks we trained for each trace set (Table 4), we ignored those larger than 25.

In summary, NeuralLTL$_f$ produced formulas that were only slightly more accurate than PMAX-SAT, and both NeuralLTL$_f$ and PMAX-SAT were unaffected by noise. However, NeuralLTL$_f$ produced significantly more zero-loss formulas as well as formulas that better captured the complexity of the target formulas. Camacho and McIlraith [4]'s SAT method

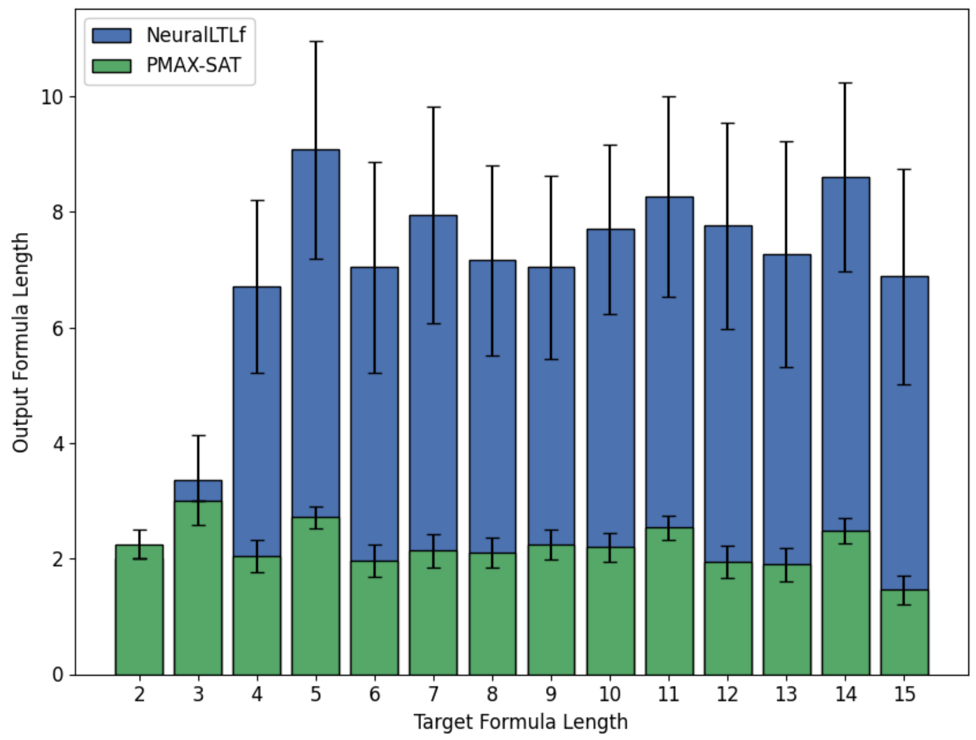

Figure 4: Output formula lengths for NeuralLTL$_f$ and PMAX-SAT on the non-noisy data with 95% confidence intervals shown.

mostly timed out on the non-noisy data and completely timed out on the noisy data.

5 Discussion

We presented two approaches to the LTL_f learning problem. First, we improved the performance of an existing SAT approach by recasting it as a PMAX-SAT problem. Our PMAX-SAT variant of Camacho and McIlraith [4]'s method produced highly accurate formulas on large and noisy trace sets where the original method timed out. However, our PMAX-SAT variant was unable to completely escape the scalability issues that arise in SAT solving. The method could only produce formulas up to length 3 in the allotted time.

NeuralLTL$_f$, on the other hand, had no problem producing larger formulas in the allotted time. The larger formulas more closely matched the complexity of the target formulas and were more accurate on the test data. However, sometimes the formulas NeuralLTL$_f$ produced were too large and overfit the data. We minimized this behavior by comparing the lengths of formulas learned by multiple network architectures on a given trace set.

Another weakness of NeuralLTL$_f$ comes from the information lost in the discretization of a continuous network during the formula extraction procedure. However, by annealing the activation functions and using random restarts, we were able to reduce the performance gap between the continuous network and discrete formula. The LTL_f formulas output by NeuralLTL$_f$ were on average only 1% less accurate than the networks they were extracted from.

Finally, the formula extraction procedure for NeuralLTL$_f$ has its own scalability issues. The temporal truth table we create during the procedure is exponentially large in the input variables (either the original propositions or the previous layer's output). Constructing and minimizing the table can be a significant bottleneck. However, the runtime of the extraction procedure does not depend on the number of traces in the dataset unlike the SAT-based methods. Additionally, as shown in our experiments, NeuralLTL$_f$ still scales to producing larger formulas than the SAT-based methods.

Reducing the expressiveness of NeuralLTL$_f$ filters would be an intriguing direction for further research. Less expressive filters would reduce the risk of overfitting with large formulas and potentially allow for a more efficient formula extraction procedure. More broadly, NeuralLTL$_f$ demonstrates a technique for modifying a neural network for the extraction of symbolic representations and forms a basis for further research in neurosymbolic artificial intelligence.

Acknowledgements

I want to thank my collaborators on this work, Daniel Ritter, Carl Trimbach, and Michael Littman. I also want to thank Michael Littman and Daniel Ritchie for serving as my advisor and reader and for providing incredible research mentorship during my time at Brown. I am super grateful for the opportunity to have worked with all of you.

References

[1] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dandelion Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. URL `https://www.tensorflow.org/`. Software available from tensorflow.org. 17

[2] Brenna D Argall, Sonia Chernova, Manuela Veloso, and Brett Browning. A survey of robot learning from demonstration. *Robotics and autonomous systems*, 57(5):469–483, 2009. 3

[3] Alper Kamil Bozkurt, Yu Wang, Michael M Zavlanos, and Miroslav Pajic. Control synthesis from linear temporal logic specifications using model-free reinforcement learning. *arXiv preprint arXiv:1909.07299*, 2019. 3

[4] Alberto Camacho and Sheila A McIlraith. Learning interpretable models expressed in linear temporal logic. In *Proceedings of the International Conference on Automated Planning and Scheduling*, volume 29, pages 621–630, 2019. 3, 4, 7, 19, 20, 22, 24

[5] Alberto Camacho, Meghyn Bienvenu, and Sheila A McIlraith. Finite LTL synthesis with environment assumptions and quality measures. *arXiv preprint arXiv:1808.10831*, 2018. 3

[6] Byungki Cha, Kazuo Iwama, Yahiko Kambayashi, and Shuichi Miyazaki. Local search algorithms for partial maxsat. *AAAI/IAAI*, 263268, 1997. 9

[7] Sreerupa Das and Michael C Mozer. A unified gradient-descent/clustering architecture for finite state machine induction. In *Advances in Neural Information Processing Systems*, pages 19–26, 1994. 4

[8] Giuseppe De Giacomo and Moshe Y Vardi. Linear temporal logic and linear dynamic logic on finite traces. In *IJCAI'13 Proceedings of the Twenty-Third International Joint Conference on Artificial Intelligence*, pages 854–860. Association for Computing Machinery, 2013. 3, 5

[9] Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In *International conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340. Springer, 2008. 20

[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. 29

[11] Kunihiko Fukushima. Neural network model for a mechanism of pattern recognition unaffected by shift in position-neocognitron. *IEICE Technical Report, A*, 62(10):658–665, 1979. 9

[12] Carlo A Furia and Paola Spoletini. On relaxing metric information in linear temporal logic. In *2011 Eighteenth International Symposium on Temporal Representation and Reasoning*, pages 72–79. IEEE, 2011. 5

[13] Mohammadhosein Hasanbeig, Yiannis Kantaros, Alessandro Abate, Daniel Kroening, George J Pappas, and Insup Lee. Reinforcement learning for temporal logic control synthesis with probabilistic satisfaction guarantees. In *2019 IEEE 58th Conference on Decision and Control (CDC)*, pages 5338–5343. IEEE, 2019. 3

[14] Joseph Kim, Christian Muise, Ankit Shah, Shubham Agarwal, and Julie Shah. Bayesian inference of linear temporal logic specifications for contrastive explanations. In *IJCAI*, pages 5591–5598, 2019. 4

[15] Diederik P. Kingma and Jimmy Ba. Adam: A Method for Stochastic Optimization. In *ICLR 2015*, 2015. 17

[16] Caroline Lemieux, Dennis Park, and Ivan Beschastnikh. General ltl specification mining (t). In *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 81–92. IEEE, 2015. 3

[17] Xiao Li, Cristian-Ioan Vasile, and Calin Belta. Reinforcement learning with temporal logic rewards. In *2017 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 3834–3839. IEEE, 2017. 3

[18] Michael L Littman, Ufuk Topcu, Jie Fu, Charles Isbell, Min Wen, and James MacGlashan. Environment-independent task specifications via GLTL. *arXiv preprint arXiv:1704.04341*, 2017. 3

[19] Joshua J Michalenko, Ameesh Shah, Abhinav Verma, Richard G Baraniuk, Swarat Chaudhuri, and Ankit B Patel. Representing formal languages: A comparison between

finite automata and recurrent neural networks. *arXiv preprint arXiv:1902.10297*, 2019. 4

[20] Daniel Neider and Ivan Gavran. Learning linear temporal properties. In *2018 Formal Methods in Computer Aided Design (FMCAD)*, pages 1–10. IEEE, 2018. 3, 4

[21] Christian W Omlin and C Lee Giles. Extraction of rules from discrete-time recurrent neural networks. *Neural Networks*, 9(1):41–52, 1996. 4

[22] Doron Peled and Thomas Wilke. Stutter-invariant temporal properties are expressible without the next-time operator. *Information Processing Letters*, 63(5):243–246, 1997. 19, 29

[23] Amir Pnueli. The temporal logic of programs. In *18th Annual Symposium on Foundations of Computer Science (sfcs 1977)*, pages 46–57. IEEE, 1977. 5

[24] Wolfgang Rautenberg. *A concise introduction to mathematical logic*. Springer, 2010. 12, 16

[25] Richard L Rudell. Multiple-valued logic minimization for PLA synthesis. Technical report, California University Berkeley Electronics Research Lab, 1986. 17

[26] Moshe Y Vardi. Alternating automata: Unifying truth and validity checking for temporal logics. In *International Conference on Automated Deduction*, pages 191–206. Springer, 1997. 7

[27] Homer Walke, Daniel Ritter, Carl Trimbach, and Michael Littman. Learning finite linear temporal logic specifications with a specialized neural operator. Under Review, 2021. 2

[28] Gail Weiss, Yoav Goldberg, and Eran Yahav. Extracting automata from recurrent neural networks using queries and counterexamples. In *International Conference on Machine Learning*, pages 5247–5256. PMLR, 2018. 4

6 Appendix

Here we provide a proof that using Spot in our experiments was valid.

6.1 Using Spot for LTL_f Simplification

Spot is a library we use to simplify formulas extracted from NeuralLTL_f networks [10]. However, Spot is designed for LTL not LTL_f, so its use requires justification. We show any simplication rule that is valid for *qualitative* LTL is also valid for *qualitative* LTL_f. Since we only test qualitative formulas in our experiments, using Spot for simplification does not introduce any invalid simplifications.

Given a trace π, π_t is the truth assignment at timestep t and $\overline{\pi_t}$ denotes that a timestep is repeated infinitely. We first prove the following useful lemma.

Lemma 1. *Take a finite trace $\pi^f = \pi_0...\pi_n$ and repeat the last timestep to create an infinite trace, $\pi^i = \pi_0...\pi_{n-1}\overline{\pi_n}$. Then given a qualitative formula ϕ, π^f satisfies ϕ interpreted as LTL_f if and only if π^i satisfies ϕ interpreted as LTL. That is, $\pi_0...\pi_n \models \phi \iff \pi_0...\pi_{n-1}\overline{\pi_n} \models \phi$.*

Proof. Qualitative LTL and LTL_f formulas are stutter-invariant, so repeating timesteps or removing timesteps does not change the truth value of a trace [22]. □

Theorem 3. *All qualitative LTL rewritings are also valid LTL_f rewritings.*

Proof. Consider the qualitative LTL rewriting $\phi \equiv \psi$. That is, for infinite traces $\pi^i \models \phi \iff \pi^i \models \psi$. We want to show for LTL_f on finite traces $\pi^f \models \phi \iff \pi^f \models \psi$.

Take a finite trace $\pi^f = \pi_0...\pi_n$. By Lemma 1, if $\pi_0...\pi_n \models \phi$, then the infinite trace $\pi_0...\pi_{n-1}\overline{\pi_n} \models \phi$. Then because $\phi \equiv \psi$, we have $\pi_0...\pi_{n-1}\overline{\pi_n} \models \psi$. Again by Lemma 1, the finite trace $\pi_0...\pi_n \models \psi$. Thus, for every finite trace $\pi^f \models \phi \implies \pi^f \models \psi$. The same argument applies to show $\pi^f \models \psi \implies \pi^f \models \phi$. So $\pi^f \models \psi \iff \pi^f \models \phi$ and the rewriting $\phi \equiv \psi$ is valid for LTL_f. □

Because all qualitative LTL rewritings are also valid LTL_f rewritings, our use of Spot is valid.