Back-propagation through Signal Temporal Logic Specifications: Infusing Logical Structure into Gradient-Based Methods

Karen Leung[1], Nikos Arechiga[2], and Marco Pavone[1]*

[1] Stanford University, Stanford, CA, USA
{karenl7, pavone}@stanford.edu
[2] Toyota Research Institute, Los Altos, CA, USA
nikos.arechiga@tri.global

Abstract. This paper presents a technique, named `stlcg`, to compute the quantitative semantics of Signal Temporal Logic (STL) formulas using computation graphs. This provides a platform which enables the incorporation of logic-based specifications into robotics problems that benefit from gradient-based solutions. Specifically, STL is a powerful and expressive formal language that can specify spatial and temporal properties of signals generated by both continuous and hybrid systems. The quantitative semantics of STL provide a robustness metric, i.e., how much a signal satisfies or violates an STL specification. In this work we devise a systematic methodology for translating STL robustness formulas into computation graphs. With this representation, and by leveraging off-the-shelf auto-differentiation tools, we are able to back-propagate through STL robustness formulas and hence enable a natural and easy-to-use integration with many gradient-based approaches used in robotics. We demonstrate, through examples stemming from various robotics applications, that `stlcg` is versatile, computationally efficient, and capable of injecting human-domain knowledge into the problem formulation.

Keywords: Signal temporal logic, robustness, computation graph, back-propagation, logical structure

1 Introduction

Many problems in robotics have solution methods that rely heavily on the availability of gradients for their computational efficiency. For instance, model predictive control often relies on solving optimization problems via gradient-based methods at each iteration, and a core mechanism behind deep learning is the ability to back-propagate through neural networks in order to perform gradient descent. On the other hand, introducing logical structure into the problem, such as through task specifications (e.g., a robot must do task A before it does task B, or a robot must stay inside a region for ten time steps before moving on)

* This work was supported by the Office of Naval Research (Grant N00014- 17-1-2433) and by the Toyota Research Institute ("TRI"). This article solely reflects the opinions and conclusions of its authors and not of ONR, TRI or any other Toyota entity.

or encoding human-domain knowledge into the model (e.g., cars must stop at a stop sign before crossing an intersection) is often desirable as this imposes a necessary requirement for a robot to operate in the environment. However, adding such discrete notions into the problem causes many gradient-based methods to become ineffective, and can significantly increase the overall problem complexity as this may introduce a combinatorial search element to the problem.

Such logic-based specifications can be expressed using temporal logic, a language that provides rules and formalisms for representing and reasoning about propositions qualified in terms of time. More specifically, Signal Temporal Logic (STL) [1, 2] is a temporal logic that is specified over dense-time real-valued signals, such as time-series data produced from continuous and hybrid systems prevalent in many robotics applications. STL provides a concise language to construct specifications (i.e., formulas) that describe relationships between the spatial and temporal properties of a signal (e.g., states of a robotic system), and can determine whether that specification is true or false. Furthermore, STL is equipped with *quantitative semantics* which provides the *robustness* value of the specification for a given signal, i.e., a continuous real-value that measures the degree of satisfaction or violation. Accordingly, STL is attractive in that it can provide a concise language for logic-based specifications needed in many problems in the field of robotics, and also the quantitative properties necessary for many existing gradient-based methods.

Our work addresses the disconnect between gradient-based techniques and logical structure by evaluating STL robustness formulas using computation graphs. The key novelty of this work stems from devising a systematic methodology to translate STL formulas into the same computational language as gradient-based methods and thus we are able to bridge together logical structure stemming from STL with many different types of problems in robotics. We develop a technique that uses computation graphs to represent STL specifications and thereby (i) inherit many of the advantages of using computation graphs, such as their computational efficiency, easy access to gradient information, and connections with deep learning software, and (ii) provide an easy-to-use framework that incorporates STL formalisms into existing methodologies. This computational paradigm connects STL (and other similar temporal logic languages) to many other fields that can benefit from spatial and temporal structure (e.g., deep learning), and hence provide desired logical structure and robustness to the problem.

Related work: In recent years, STL (and other similar temporal logic languages such as Metric Temporal Logic (MTL) [3] and Truncated Linear Temporal Logic (TLTL) [4]) has been used in robotics to ensure that the system satisfies desired logic-based constraints. Examples include model predictive control [3, 5], stochastic control [6–8], and learning-based control [9] problems to ensure the robot trajectory satisfies a set of specifications. Temporal logic has also been used in reinforcement learning settings to learn policies that enable the robot to achieve long-term goals [4, 10, 11], and for extracting features from time-series data for behavioral clustering [12, 13].

One of the main challenges, however, is in developing a computational framework for STL that is efficient, tractable, and flexible such that it can be easily integrated within a variety of existing frameworks in robotics. For example, model predictive control for continuous time systems with STL specifications can be reformulated as a Mixed-Integer Linear Program (MILP) [5, 6]. These methods are shown to be successful in some case studies, though a main limitation is that they are difficult to scale to large and complex systems. Solving MILPs is NP-hard and therefore solution computations do not scale well with problem size. On the other hand, [3] applies smooth approximations on the robustness formulas and uses sequential quadratic programming (SQP) to carry out the optimization. They show that their method outperforms [5] in terms of computation time, but their formulation is targeted at SQPs only.

In contrast, our proposed computation graph representation of STL robustness formulas, `stlcg`, is lightweight and agnostic to the application. As a result, it can be combined with a variety of existing gradient-based methods to provide desired logical structure without significant compromises to problem tractability. With the case studies investigated in this work, we demonstrate how `stlcg` can be used to embed logical structure within gradient-based optimization algorithms, such as deep neural networks and trajectory optimization problems, where the output obeys desired STL specifications. We also show for an established class of problems making use of STL that `stlcg` lends itself readily to parallelization for computational speed. Further, our technique provides an alternative approach aimed towards verifying deep neural networks, i.e., quantifying the output space of a network. Current techniques for neural network verification suffer from scalability issues or are limited to a subset of activation functions (see [14] for a survey on neural network verification methods). We envision that by embedding logical structure upstream of the learning process, we can help improve robustness and regularity of the model, and thus enhance performance in downstream applications.

Statement of contributions: The contributions of this paper are threefold. First, we describe the mechanism behind `stlcg` by detailing the construction of the computation graph for an arbitrary STL formula. Through this construction, we prove the correctness of `stlcg`, and show that it scales at most quadratically with input length and linearly in formula size. The translation of STL formalisms into computation graphs allows `stlcg` to inherit many benefits such as the ability to back-propagate through the graph to obtain gradient information; this abstraction also provides computational benefits such as enabling portability between hardware backends (e.g., CPU and GPU) and batching for scalability. Second, we open-source our PyTorch implementation of `stlcg`; it is a toolbox that makes `stlcg` easy to combine with many existing frameworks and deep learning models. Third, we highlight key advantages of `stlcg` by investigating a number of diverse example applications in robotics such as motion planning, logic-based parameter fitting, regression, and intent prediction. We emphasize `stlcg`'s computational efficiency and natural ability to embed logical specifications into the problem formulation to make the output more robust.

Organization: In Section 2, we cover the basic definitions and properties of STL, and describe the graphical structure of STL formulas. Equipped with the graphical structure of an STL formula, Section 3 draws connections to computation graphs and provides details on the technique that underpins `stlcg`. We showcase the benefits of `stlcg` through a variety of case studies in Section 4.

We finally conclude in Section 5 and propose exciting future research directions for this work.

2 Preliminaries

In this section, we provide the definitions and syntax for STL, and describe the underlying graphical structure of the computation graph used to evaluate STL robustness formulas.

2.1 Signals

STL formulas are interpreted over *signals*, which we define formally as follows.

Definition 1 (Signal). *A signal $s_{t_0} = (x_0, t_0), (x_1, t_1), \ldots, (x_T, t_T)$ is an ordered ($t_{i-1} < t_i$) finite sequence of states $x_i \in \mathbb{R}^n$ and their associated times $t_i \in \mathbb{R}$. For ease of notation, s (i.e., when the subscript on s_t is dropped) denotes the entire signal.*

A signal represents real-valued, discrete-time output (i.e., continuous-time output sampled at finite time intervals) from any system of interest, such as a sequence of robot states, the temperature of a building, or the speed of a vehicle. In this work, we assume that the signal is sampled at uniform time steps Δt. Further, we define a *subsignal*.

Definition 2 (Subsignal). *Given a signal s_{t_0}, a subsignal s_{t_i} is also a signal where $i \geq 0$, $t_i \geq t_0$, and $s_{t_i} = (x_i, t_i), (x_{i+1}, t_{i+1}), \ldots, (x_T, t_T)$.*

2.2 Signal Temporal Logic: Syntax and Semantics

STL formulas are defined recursively according to the following grammar [1, 2],

$$\phi ::= \top \mid \mu_c \mid \neg\phi \mid \phi \wedge \psi \mid \phi\, \mathcal{U}_{[a,b]}\, \psi. \tag{1}$$

$\top$ means true, μ_c is a predicate of the form $\mu(x) > c$ where $c \in \mathbb{R}$ and $\mu : \mathbb{R}^n \to \mathbb{R}$ is a differentiable function that maps the state $x \in \mathbb{R}^n$ to a scalar value, ϕ and ψ are STL formulas, and $[a, b] \subseteq \mathbb{R}_{\geq 0}$ is a time interval. When the time interval is omitted, the temporal operator is evaluated over the positive ray $[0, \infty)$. The symbols $\neg$ (negation/not), and $\wedge$ (conjunction/and) are logical connectives, and $\mathcal{U}$ (until) is a temporal operator. Additionally, other commonly used logical connectives ($\vee$ (disjunction/or) and $\Rightarrow$ (implies)), and temporal operators ($\lozenge$ (eventually), and $\square$ (always)) can be defined as follows,

$$\phi \vee \psi = \neg(\neg\phi \wedge \neg\psi), \quad \phi \Rightarrow \psi = \neg\phi \vee \psi,$$
$$\lozenge_{[a,b]}\, \phi = \top\, \mathcal{U}_{[a,b]}\, \phi, \quad \square_{[a,b]}\, \phi = \neg\lozenge_{[a,b]}(\neg\phi).$$

STL formulas are constructed by taking a predicate μ_c and then recursively applying a logical connective or temporal operator to it (see Example 1).

Example 1. Let $x \in \mathbb{R}^n$, $\phi_1 = \mu_1(x) < c_1$ and $\phi_2 = \mu_2(x) \geq c_2$.[3] Then, for a given signal s_t, the formula $\phi_3 = \phi_1 \wedge \phi_2$ is true if both ϕ_1 and ϕ_2 are true. Similarly, $\phi_4 = \square_{[a,b]}\phi_3$ is true if ϕ_3 is true over the entire interval $[t + a, t + b]$.

[3] Equality and the other inequality relations can be derived from the STL grammar in (1), i.e., $\mu(x) < c \Leftrightarrow -\mu(x) > -c$, and $\mu(x) = c \Leftrightarrow \neg(\mu(x) < c) \wedge \neg(\mu(x) > c)$.

We use the notation $s_t \models \phi$ to denote that a signal s_t satisfies an STL formula ϕ according to the formal semantics below. Informally, $s_t \models \phi \,\mathcal{U}_{[a,b]}\, \psi$ if over the time interval $[t+a, t+b]$, ϕ holds for all time before ψ holds, $s_t \models \lozenge_{[a,b]}\, \phi$ if at some time $t \in [t+a, t+b]$, ϕ holds at least once, and $s_t \models \square_{[a,b]}\, \phi$ if ϕ holds for all $t \in [t+a, t+b]$. Formally, the *Boolean* semantics (i.e., if it is true or false) of a formula with respect to a signal s_t is defined inductively as follows.

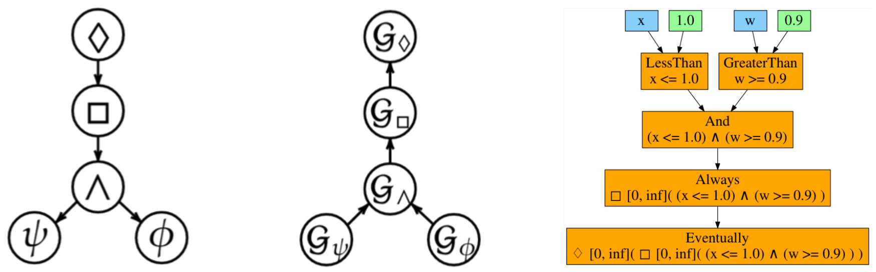
$$\begin{array}{lcl}
s_t \models \mu_c & \Leftrightarrow & \mu(x_t) > c \\
s_t \models \neg\phi & \Leftrightarrow & \neg(s_t \models \phi) \\
s_t \models \phi \wedge \psi & \Leftrightarrow & (s_t \models \phi) \wedge (s_t \models \psi) \\
s_t \models \phi \vee \psi & \Leftrightarrow & (s_t \models \phi) \vee (s_t \models \psi) \\
s_t \models \phi \Rightarrow \psi & \Leftrightarrow & \neg(s_t \models \phi) \vee (s_t \models \psi) \\
s_t \models \lozenge_{[a,b]}\phi & \Leftrightarrow & \exists t' \in [t+a, t+b] \;\text{ s.t. } s_{t'} \models \phi \\
s_t \models \square_{[a,b]}\phi & \Leftrightarrow & \forall t' \in [t+a, t+b] \;\; s.t. \; s_{t'} \models \phi \\
s_t \models \phi \,\mathcal{U}_{[a,b]}\, \psi & \Leftrightarrow & \exists t' \in [t+a, t+b] \;\; s.t. \; (s_{t'} \models \psi) \;\wedge\; (s_t \models \square_{[0,t']}\phi)
\end{array}$$

Furthermore, STL admits the notion of *robustness*, that is, it has *quantitative semantics* that calculates the degree of satisfaction or violation a signal has for a given formula. Positive robustness values indicate satisfaction, while negative robustness values indicate violation. Like Boolean semantics, the quantitative semantics of a formula with respect to a signal s_t is defined inductively as follows, where $\rho_{max} > 0$.

$$\begin{array}{rcl}
\rho(s_t, \top) & = & \rho_{\max} \\
\rho(s_t, \mu_c) & = & \mu(x_t) - c \\
\rho(s_t, \neg\phi) & = & -\rho(s_t, \phi) \\
\rho(s_t, \phi \wedge \psi) & = & \min(\rho(s_t, \phi), \rho(s_t, \psi)) \\
\rho(s_t, \phi \vee \psi) & = & \max(\rho(s_t, \phi), \rho(s_t, \psi)) \\
\rho(s_t, \phi \Rightarrow \psi) & = & \max(-\rho(s_t, \phi), \rho(s_t, \psi)) \\
\rho(s_t, \lozenge_{[a,b]}\phi) & = & \max\limits_{t' \in [t+a, t+b]} \rho(s_{t'}, \phi) \\
\rho(s_t, \square_{[a,b]}\phi) & = & \min\limits_{t' \in [t+a, t+b]} \rho(s_{t'}, \phi) \\
\rho(s_t, \phi \,\mathcal{U}_{[a,b]}\, \psi) & = & \max\limits_{t' \in [t+a, t+b]} (\min(\rho(s_{t'}, \psi), \min\limits_{t'' \in [0,t']} \rho(s_{t''}, \phi)))
\end{array}$$

Further, we define the *robustness trace* as a sequence of robustness values of every subsignal s_{t_i} of signal s_{t_0}, $t_i \geq t_0$.

Definition 3 (Robustness trace). *Given a signal s_{t_0} and an STL formula ϕ, the robustness trace $\tau(s_{t_0}, \phi)$ is a finite sequence of robustness values of ϕ for each subsignal of s_{t_0}. Specifically, $\tau(s_{t_0}, \phi) = \rho(s_{t_0}, \phi), \rho(s_{t_1}, \phi), \ldots, \rho(s_{t_T}, \phi)$ where $t_{i-1} < t_i$.*


x
1.0
w
0.9
LessThan
x <= 1.0
GreaterThan
w >= 0.9
And
(x <= 1.0) ∧ (w >= 0.9)
Always
□ [0, inf]((x <= 1.0) ∧ (w >= 0.9))
Eventually
◇ [0, inf](□ [0, inf]((x <= 1.0) ∧ (w >= 0.9)))

(a) Parse tree $\mathcal{T}$. (b) Computation graph $\mathcal{G}$. (c) Visualization of $\mathcal{G}$ using the `stlcg` toolbox.

Fig. 1: An illustration of a parse tree (left) and a computation graph (middle) for an STL formula $\lozenge \square (\phi \wedge \psi)$ where $\phi = w \geq 0.9$ and $\psi = x \leq 1.0$. On the right is a visualization generated from the `stlcg` toolbox. The blue nodes represents input signals (x and w) into the computation graph, the green nodes represent parameters of the predicates, and the orange nodes represent STL operations.

2.3 Graphical Structure of STL Formulas

STL formulas are defined recursively according to the grammar in (1). We can leverage the recursive structure to represent an STL formula with a parse tree $\mathcal{T}$ by identifying its subformulas. A *subformula* of an STL operator is a formula which the operator is applied to. The operators $\wedge$, $\vee$, $\Rightarrow$, and $\mathcal{U}$ will have two corresponding subformulas, while predicates are not defined recursively and thus have no subformulas. For example, the subformula of $\square \phi$ is ϕ, and the subformulas of $\phi \wedge \psi$ are ϕ and ψ. The root node of a parse tree corresponds to the outer-most operator for the formula. This node is connected to the outer-most operator of its subformulas, and so forth. Applying this recursion, each node represents each operation that makes up the STL formula, starting from the outer-most operator as the root node and ending with the inner-most operators as the leaf nodes (i.e., the leaf nodes represent predicates). An example of a parse tree for the formula $\lozenge \square (\phi \wedge \psi)$ is illustrated in Figure 1a.

By flipping the direction of the edges in $\mathcal{T}$, we obtain a directed acyclic graph $\mathcal{G}$, such as the one shown in Figure 1b. $\mathcal{G}$ represents the structure of the STL computation graph to be detailed in Section 3. At a high level, signals are passed through the root nodes ($\mathcal{G}_\phi$ and $\mathcal{G}_\psi$ in Figure 1b) to produce robustness traces from applying those STL operations. The output robustness traces are then passed through the next node of the graph ($\mathcal{G}_\wedge$ in Figure 1b) to produce the robustness trace from applying that STL operation, and so forth.

3 STL Robustness Formulas as Computation Graphs

We first describe how to represent each STL operator as a computation graph, and then show how to combine them together to form the overall computation graph $\mathcal{CG}$ that computes the robustness trace of any given STL formula. Further, we provide smooth approximations to the max and min operations in order to help make the gradients smoother when back-propagating through the graph,

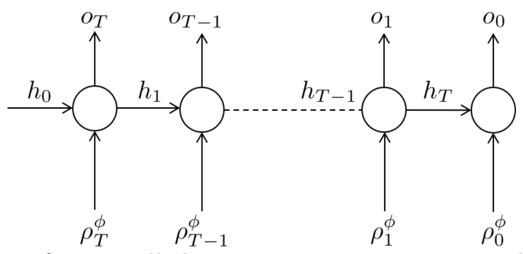


Fig. 2: A schematic of an unrolled recurrent computation graph for the $\Diamond$ (eventually) and $\Box$ (always) temporal operators (differing in the implementation of the recurrent cells depicted as circles).

and introduce a new temporal operator which addresses some limitations of using the max and min functions. The resulting computational framework, `stlcg` (the code can be found at `https://github.com/StanfordASL/stlcg`), is implemented using PyTorch [15]. Further, this toolbox includes a graph visualizer, illustrated in Figure 1c, to show the graphical representation of the STL formula and how it depends on the inputs and parameters from the predicates.

3.1 Computation Graphs

A computation graph is a directed graph where the nodes correspond to operations or variables. Values of the variable are fed into operations, and the outputs can feed into other operations. When the operations are differentiable, we can back-propagate through a computation graph and obtain gradients with respect to any variable inside the computation graph. This is the underlying mechanism behind automatic differentiation software which are designed to make back-propagation computationally efficient.

3.2 Computation Graphs of STL Operators

First we consider the computation graph of each STL robustness formula given in Section 2.2 individually. The formulas for the non-temporal operators are relatively straightforward to implement as computation graphs involving simple mathematical operations, such as subtraction, max, and min (i.e., the operation is applied over each element of the input robustness trace(s)). To compute robustness for the temporal operators, specifically $\Diamond_{[a,b]}$, $\Box_{[a,b]}$, and $\mathcal{U}_{[a,b]}$, we apply dynamic programming [16] by using a recurrent computation graph, similar to the structure of a recurrent neural network (RNN) [17]. The complexity of this approach is linear in the length of the signal for $\Diamond$ and $\Box$, and quadratic for the $\mathcal{U}$ operator, and linear in the number of nodes in the parse tree.

We first consider the $\Diamond$ operator, noting that similar constructions apply for $\Box$ and $\mathcal{U}$. Let $\psi = \Diamond_{[a,b]}\phi$; the goal is to construct the computation graph $\mathcal{G}_{\Diamond_{[a,b]}}$ which applies the $\Diamond_{[a,b]}$ operation over the input signal $\tau(s_{t_0}, \phi)$ (the robustness trace of the subformula). For ease of notation, we denote $\rho(s_{t_i}, \phi) = \rho_i^\phi$ as the robustness of ϕ for subsignal s_{t_i}. To apply the dynamic programming recursion, the robustness trace is fed into the computation graph *backwards* in time.

Figure 2 illustrates the unrolled graphical structure of $\mathcal{G}_{\Diamond_{[a,b]}}$. The ith recurrent node takes in a hidden state h_i and an input state ρ_{T-i}^ϕ, and produces an output state o_{T-i} (note the time indices as the input robustness trace is reversed). By collecting all the o_i's, the output of the computation graph is the (backwards) robustness trace of ψ, i.e., $\tau(s_{t_0}, \psi)$. The output robustness trace is treated as the input to another computation graph representing the next STL operator dictated by $\mathcal{G}$.

Before describing the construction of the computation graph $\mathcal{CG}$, we first define $M_N \in \mathbb{R}^{N\times N}$ to be a square matrix with ones on the upper off-diagonal and $B_N \in \mathbb{R}^N$ to be a vector of zeros with a one in the last entry. If $x \in \mathbb{R}^N$ and $u \in \mathbb{R}$, then the operation $M_N x + B_N u$ removes the first element of x, shifts all the entries up one index and replaces the last entry with u. We distinguish four cases depending on the interval attached to the temporal operator.

Case 1: $[0, \infty)$ Let the initial hidden state be $h_0 = \rho^{\phi}_T$.[4] The output for the first step is $o_T = \max(h_0, \rho^{\phi}_T)$, and the next hidden state is defined to be $h_1 = o_T$. Hence the general update step becomes, $h_{i+1} = o_{T-i}$, and $o_{T-i} = \max(h_i, \rho^{\phi}_{T-i})$. By construction, the last step in this dynamic programming step is,

$$\begin{aligned} o_0 &= \max(h_T, \rho^{\phi}_0) \\ o_0 &= \max(h_0, \rho^{\phi}_T, \rho^{\phi}_{T-1}, \ldots, \rho^{\phi}_0) \\ o_0 &= \rho(s, \Diamond\phi) = \rho(s, \psi). \end{aligned}$$

The output of $\mathcal{G}_{\Diamond_{[0,\infty)}\phi}$, $o_0, o_1, \ldots, o_T$, is the robustness trace of $\psi = \Diamond_{[0,\infty)}\phi$, and the last output o_0 is $\rho(s, \psi)$.

Case 2: $[0, b]$**,** $b < \infty$ Let the initial hidden state be $h_0 = (h_{0_1}, h_{0_2}, \ldots, h_{0_{N-1}})$, where $h_{0_i} = \rho^{\phi}_T$, $\forall i = 1, ..., N-1$ and N is the number of time steps contained in the interval $[0, b]$. The hidden state is designed to keep track of inputs in the interval $(0, b]$. The output for the first step is $o_T = \max(h_0, \rho^{\phi}_T)$ (the max operation is over the elements in h_0 and ρ^{ϕ}_T), and the next hidden state is defined to be $h_1 = M_{N-1}h_0 + B_{N-1}\rho^{\phi}_T$. Hence the general update step becomes,

$$h_{i+1} = M_{N-1}h_i + B_{N-1}\rho^{\phi}_{T-i}, \qquad o_{T-i} = \max(h_i, \rho^{\phi}_{T-i}).$$

By construction, o_0 corresponds to the definition of $\rho(s, \Diamond_{[0,b]}\phi)$, $b < \infty$.

Case 3: $[a, \infty)$**,** $a > 0$ Let the hidden state be a tuple $h_i = (c_i, d_i)$. Let the initial hidden state be $h_0 = (\rho^{\phi}_T, d_0)$ where $d_0 = (d_{0_1}, d_{0_2}, \ldots, d_{0_{N-1}})$, $d_{0_i} = \rho^{\phi}_T$, $\forall i = 1, ..., N-1$ and N is the number of time steps encompassed in the interval $[0, a]$. The output for the first step is $o_T = \max(c_0, d_{0_1})$, and the next hidden state is defined to be $h_1 = (o_T, M_{N-1}d_0 + B_{N-1}\rho^{\phi}_T)$. Following this pattern, the general update step becomes,

$$h_{i+1} = (o_{T-i}, M_{N-1}d_i + B_{N-1}\rho^{\phi}_{T-i}), \qquad o_{T-i} = \max(c_i, d_{i_1}).$$

By construction, o_0 corresponds to the definition of $\rho(s, \Diamond_{[a,\infty)}\phi)$, $a > 0$.

Case 4: $[a, b]$**,** $a > 0$, $b < \infty$ We combine the ideas from Cases 2 and 3. Let the initial hidden state be $h_0 = (h_{0_1}, h_{0_2}, \ldots, h_{0_{N-1}})$, where $h_{0_i} = \rho^{\phi}_T$, $\forall i =$

[4] This corresponds to padding the input trace with the last value of the robustness trace. A different value could be chosen instead. This applies to Cases 2-4 as well.

	h_0	o_5	h_1	o_4	h_2	o_3	h_3	o_2	h_4	o_1	h_5	o_0
$\Diamond_{[0,\infty)}(s>0)$	1	1	1	3	3	3	3	3	3	3	3	3
$\Diamond_{[0,2]}(s>0)$	(1, 1)	1	(1, 1)	3	(1, 3)	3	(3, 2)	3	(2, 1)	2	(1, 1)	1
$\Diamond_{[2,\infty)}(s>0)$	(1,(1,1))	1	(1,(1,1))	1	(1,(1,3))	1	(1,(3,2))	3	(3,(2,1))	3	(3,(1,1))	3
$\Diamond_{[1,3]}(s>0)$	(1, 1, 1)	1	(1, 1, 1)	1	(1, 1, 3)	3	(1, 3, 2)	3	(3, 2, 1)	3	(2, 1, 1)	2

Table 1: Hidden and output states of $\Diamond_{\mathrm{I}}(s>0)$ for a signal $s = 1, 1, 1, 2, 3, 1$ and different intervals I. The signal is fed into the computation graph *backwards*.

$1, ..., N-1$ and N is the number of time steps encompassed in the interval $[0, b]$. Let M be the number of time steps encompassed in the $[a, b]$ interval. The output for the first step is $o_T = \max(h_{0_1}, h_{0_2}, \ldots, h_{0_M})$. The next hidden state is defined to be $h_1 = M_{N-1}h_0 + B_{N-1}\rho^{\phi}_{T}$. Hence the general update step becomes,

$$h_{i+1} = M_{N-1}h_i + B_{N-1}\rho^{\phi}_{T-i}, \qquad o_{T-i} = \max(h_{i_1}, h_{i_2}, \ldots, h_{i_M})$$

By construction, i.e., inputting the robustness trace backwards, and the choice of the hidden state, and output state, we are able to compute the robustness trace for the $\Diamond$ operator in linear time. See Example 2 for a concrete example. The computation graph for $\Box$ is the same but instead uses the min operation instead of max . For brevity, we omit the construction of the computation graph for $\phi\,\mathcal{U}_{[a,b]},\psi$, but a sketch of the construction is as follows. It involves iterating over each time step of the signal and computing the robustness trace of $\Box\,\phi$ for the front section of the signal, and the robustness trace of ψ for the latter section for each iteration. The complexity for computing the robustness trace for the $\mathcal{U}$ operator grows quadratically with the length of the signal.

Example 2. Let the values of a signal be $s = 1, 1, 1, 2, 3, 1$. Then the hidden and output states for $\Diamond_I$ with different intervals I are given in Table 1.

3.3 Calculating Robustness with Computation Graphs

Given an STL formula ϕ, we can construct the computation graph corresponding to each operation defining ϕ. Since by construction, each component takes a robustness trace as its input, and outputs the robustness trace with the corresponding STL operation applied to it, we can "stack" each computation graph by following the graphical structure dictated by $\mathcal{G}$ for a given formula ϕ (refer to Figure 1b). The overall computation graph $\mathcal{CG}$ for the STL formula ϕ, which we will denote by $\mathcal{CG}_\phi$, takes a signal s_t as its input, and outputs the robustness trace $\tau(s_t, \phi)$ as its output. By construction, the robustness trace generated by the computation graph matches exactly the true robustness trace of the formula, and thus `stlcg` is correct by construction (see Theorem 1).

Theorem 1 (Correctness of `stlcg`). *For any STL formula ϕ and any signal s_t, let $\mathcal{CG}_\phi$ be the computation graph produced by `stlcg`. Then passing a signal s_t through $\mathcal{CG}_\phi$ produces the robustness trace $\tau(s_t, \phi)$.*

3.4 Smooth Approximations to `stlcg`

Due to the recursion over max and min operations, there is the potential in practice for the gradients to vanish. To mitigate this, we can leverage smooth approximations to the max and min functions. Let $x \in \mathbb{R}^n$ and $w \in \mathbb{R}_{\geq 0}$, then the max and min approximations are

$$\widetilde{\max}(x; w) = \frac{\sum_i^n x_i \exp(w x_i)}{\sum_j^n \exp(w x_j)}, \qquad \widetilde{\min}(x; w) = \frac{\sum_i^n x_i \exp(-w x_i)}{\sum_j^n \exp(-w x_j)},$$

where x_i represents the i-th element of x, and $w \geq 0$ operates as a scaling parameter. The approximation approaches the true solution when $w \to \infty$ while $w = 0$ results in the mean value of the entries of x. In practice, w can be annealed over gradient-descent iterations.

Further, max and min are pointwise functions. As a result, the robustness of an STL formula can be highly sensitive to one single point in the signal, and may not provide an adequate robustness metric especially if the signal is noisy [18]. We propose using an integral-based STL robustness formula $\mathcal{I}^M_{[a,b]}$ as an alternative to the Always robustness formula, and it is defined as follows. For a given weighting function $M(t)$,

$$\rho(s_t, \mathcal{I}^M_{[a,b]} \phi) = \sum_{\tau = t+a}^{t+b} M(\tau) \rho(s_\tau, \phi).$$

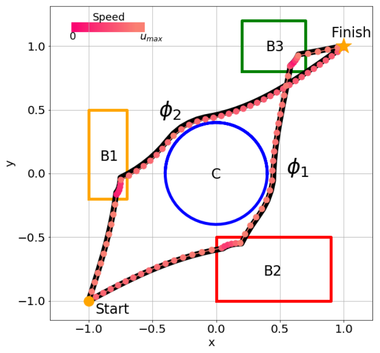
The Integral robustness formula considers the weighted sum of the input signal over an interval $[a, b]$. In contrast to the Always robustness formula which uses min, a pointwise function, the Integral operator can produce a smoother robustness trace and this behavior is demonstrated in Section 4.1.

4 Case Studies: Using `stlcg` for Robotics Applications

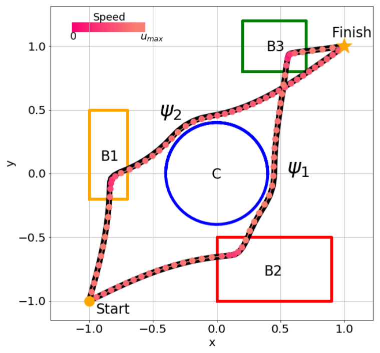
We demonstrate the versatility and computational advantages of using `stlcg` by investigating a number of case studies in this section. In these examples, we show (i) how our approach can be used to incorporate logic-based requirements into motion planning problems (Section 4.1), (ii) the computational efficiency of `stlcg` achieved via parallelization (Section 4.2), and (iii) how we can use `stlcg` to translate human-domain knowledge into a form that can be integrated with deep neural networks (Section 4.3). Code and additional figures can be found at `https://github.com/StanfordASL/stlcg`.

4.1 Motion Planning with STL Constraints

Recently, there has been a lot of interest in motion planning with STL constraints (e.g., [3, 5]); the problem of finding a sequence of states and controls that drives a robot from an initial state to a final state while obeying a set of constraints which includes STL specifications. For example, the robot may be required to enter a particular region for three time steps before moving to its final destination. Rather than reformulating the problem as a MILP to account for STL constraints as done in [5], we consider a simpler approach of augmenting the loss function



(a) Trajectories satisfying ϕ_1 and ϕ_2 which uses the Always operator.



(b) Trajectories satisfying ψ_1 and ψ_2 which uses the Integral operator.

Fig. 3: Motion planning with STL constraints solved using `stlcg`.

with terms that maximize the robustness of the STL formulas, and enable a higher degree of customization when designing how the STL specifications are formulated into the problem.

Consider the following motion planning problem illustrated in Figure 3; a robot must find a sequence of states $X = x_{1:N}$ and controls $U = u_{1:N-1}$ that takes it from the yellow circle (position = (-1,-1)) to the yellow star (position=(1,1)) while satisfying an STL constraint ϕ. Assume the robot is a point mass with state $x \in \mathbb{R}^2$ and action $u \in \mathbb{R}^2$. It obeys single integrator dynamics $\dot{x} = u$ and has a control constraint $\|u\|_2 \leq u_{\max}$. We can express the control constraint as an STL formula: $\theta = \Box \, \|u\|_2 \leq u_{\max}$. This motion planning problem can be cast as an unconstrained optimization problem,

$$\min_{X,\,U} \; \|Ez - D\|_2^2 + \gamma_1 J_m(\rho(X, \phi)) + \gamma_2 J_m(\rho(U, \theta))$$

where $z = (X, U)$ is the concatenated vector of states and controls. Since the dynamics are linear, we can express the dynamic and end point constraints across all time steps in a single linear equation $Ez = D$. J_m represents the cost function on the robustness value of the STL specifications with a margin m. We use $J_m(x) = \text{ReLU}(-(x-m))$ ($\text{ReLU}(x) = \max(0, x)$ is the rectified linear unit) which incentivizes the robustness to be greater than some margin m.

We consider four different STL constraints,

$$\begin{aligned}
\phi_1 &= \Diamond \Box_{[0,5]} \text{inside B2} \,\wedge\, \Diamond \Box_{[0,5]} \text{inside B3} \,\wedge\, \neg \Box \, \text{inside C} \\
\phi_2 &= \Diamond \Box_{[0,5]} \text{inside B1} \,\wedge\, \neg \Box \text{inside B3} \,\wedge\, \neg \Box \, \text{inside C.} \\
\psi_1 &= \Diamond \mathcal{I}_{[0,5]}^{\Delta t^{-1}} \text{inside B2} \,\wedge\, \Diamond \mathcal{I}_{[0,5]}^{\Delta t^{-1}} \text{inside B3} \,\wedge\, \neg \Box \, \text{inside C} \\
\psi_2 &= \Diamond \mathcal{I}_{[0,5]}^{\Delta t^{-1}} \text{inside B1} \,\wedge\, \neg \mathcal{I}^{\Delta t^{-1}} \text{inside B3} \,\wedge\, \neg \Box \, \text{inside C.}
\end{aligned}$$

ϕ_1 translates to: the robot needs to eventually be inside the red box (B2), and the green box (B3) for five time steps each, and never enter the blue circular

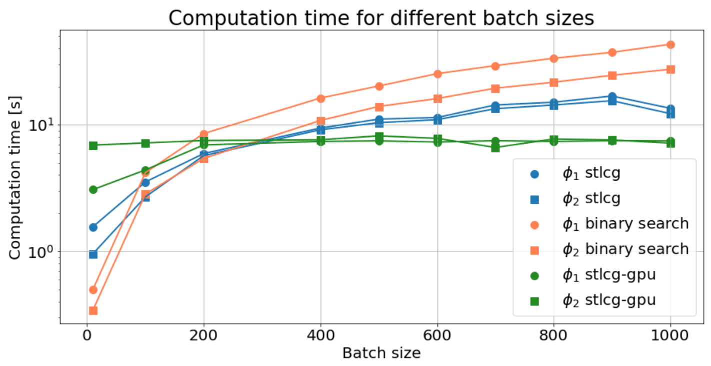


Fig. 4: Computation time of different methods used to solve a pSTL problem. This was computed using a 3.0GHz octocore AMD Ryzen 1700 CPU and a Titan X (Pascal) GPU.

region (C). Similarly, ϕ_2 translates to: the robot eventually needs to enter the orange box (B1) for five time steps, and never enter the green box (B3) nor the blue circular region (C). ψ_1 and ψ_2 are similar except that the Integral operator (described in Section 3.4) is used instead of the Always operator for the box constraints. We initialize the solution with a straight line connecting the start and end points (which violates the STL constraint), then apply gradient descent (step size of 0.05) with $\gamma_1 = \gamma_2 = 0.3$, $m = 0.05$, and $\Delta t^{-1} = \frac{1}{0.1}$; the solutions are illustrated in Figure 3. We see that, as anticipated, using the Integral operator (Figure 3b) results in a smoother trajectory than using the Always operator (Figure 3a) because the robustness value of the Integral operator depends on multiple values, while the Always operator relies only on a single point.

4.2 Parametric STL for Behavioral Clustering

In this example, we consider parametric STL (pSTL) problems [12, 13], and demonstrate that `stlcg` has the ability to parallelize the computation. The pSTL problem is a form of logic-based parameter estimation for time series data. It involves first proposing an STL template formula where the predicate parameter values are unknown and then solving for parameter values that best fit a given signal. pSTL is useful for extracting features from time-series data which can then be used for clustering. For example, pSTL has been used to cluster human driving behaviors in the context of autonomous driving applications [12].

The experimental setup is as follows. Given a dataset of step responses from randomized second-order systems, we use pSTL to cluster different types of responses, e.g., under-damped, critically damped, or over-damped. Based on domain-knowledge characterization of these properties, we design the following pSTL formulas,

$$\phi_1 = \square_{[50,100]}\, |s-1| < \epsilon_1 \quad \text{(final value)}, \qquad \phi_2 = \square\, s < \epsilon_2 \quad \text{(peak value)}.$$

For each signal $s^{(j)}$, $j = 1, \dots, N$, we want to find ϵ_{1j} and ϵ_{2j} (ϵ_1 and ϵ_2 for the jth signal) that provides the best fit, i.e., robustness equals zero. Using `stlcg`, we can *batch* our computation since each signal is decoupled from one another, and hence solve for all ϵ_{ij} simultaneously.[5] We use gradient descent to solve the optimization problem for each pSTL formula ϕ_i, $i = 1, 2$, with the following loss function, $\min_{\epsilon_{ij}} \sum_{j=1}^{N} \text{ReLU}(-\rho(s^{(j)}, \phi_i))$. In contrast, [12] proposes a binary search approach to solve monotonic pSTL formulas, formulas where the robustness value increases monotonically as the parameters increases (or decreases). However, this method can only optimize one signal at a time. As ϕ_1 and ϕ_2 are monotonic pSTL formula, we can compare the average computation time taken to find a solution to all ϵ_{ij}'s using `stlcg`[6] with the binary search approach in [12]. Both approaches converged to the same solution, and the results are illustrated in Figure 4. We see that the computation time for the binary search method increases linearly, while the computation time using `stlcg` increases at a much lower rate due to batching, but requires some initial overhead to set up the computation graph. Further, since our `stlcg` toolbox leverages the autodifferentiation tool in PyTorch, we can very easily make use of the GPU, which accelerates the learning process, and provides near constant time computation for sufficiently large problem sizes.

For cases where the solution has multiple local minima (e.g., non-monotonic pSTL formulas), we can additionally batch the input with samples from the parameter space, and anneal w, the scaling parameter for the max and min approximation over each iteration. The samples will converge to a local minimum, and, with sufficiently many samples and an adequate annealing schedule, we can (hopefully) find the global minimum. However, we note that we are currently not able to optimize time parameters that define an interval as we cannot back-propagate through those parameter. Future work will investigate how to address this, potentially leveraging ideas from forget gates in long short-term memory networks [19].

4.3 Robustness-Aware Neural Networks

In the following examples, we demonstrate how `stlcg` can be used to make learning-based models (e.g., deep neural networks) more robust and reflect desired behaviors stemming from human-domain knowledge that can be expressed as STL specifications.

Model Fitting with Known Structure Consider the problem of using a neural network to approximate, or smooth out, temporal data such as the (noisy) signal shown in Figure 5 (left). Suppose that based on domain-knowledge of the problem, we know that the data must satisfy the STL formula $\phi = \square_{[1,3]}(s > 0.48 \wedge s < 0.52)$. Due to the noise in the data (e.g., from sensor noise or poor state estimation), ϕ is violated. Neural networks are prone to over-fitting and may unintentionally learn the noise. For instance, a very simple single layer neural network f_θ could memorize the noisy signal, leading to the learned model violating ϕ (see Figure 5 (left)). To mitigate this, we regularize the learning processing by augmenting the loss function with a term that penalizes negative

[5] We can even account for variable signal length by padding the inputs and keeping track of the signal lengths.

[6] The ϵ_{ij}'s are initialized to zero, which gives negative robustness values and hence results in a non-zero gradient.

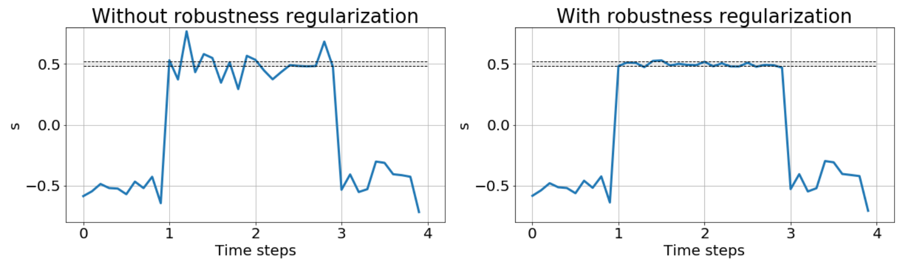


Fig. 5: A simple supervised learning problem without (left) and with (right) STL regularization. The output is required to satisfy $\phi = \square_{[1,3]}(s > 0.48 \wedge s < 0.52)$ despite the training data being noisy and violating ϕ.

robustness values. Specifically, the loss function becomes

$$\mathcal{L} = \mathcal{L}_0 + \gamma \mathcal{L}_\rho \tag{2}$$

where $\mathcal{L}_0$ is the original loss function (e.g., reconstruction loss with L2 regularization on the parameters), $\mathcal{L}_\rho$ is the robustness loss defined in this example to be $\mathcal{L}_\rho = \mathrm{ReLU}(-\rho(f_\theta(x), \phi))$, and $\gamma = 10$. The resulting model with robustness regularization is able to obey ϕ more robustly as shown in Figure 5 (right). Note that this does not guarantee that the output will satisfy ϕ but rather the resulting model is encouraged to ensure that ϕ is violated as little as possible and can be further incentivized by further increasing γ.

Sequence-to-Sequence Prediction We consider a sequence-to-sequence prediction model and demonstrate how regularizing with STL robustness formulas can result in better long-term prediction performance despite only having access to short-term data. Sequence-to-sequence prediction models are often used in robotics, such as in the context of model-based control where a robot may predict the future trajectories of other agents in the environment given past trajectories, and use these predictions for decision-making and control [20]. Often contextual knowledge of the environment is not explicitly labeled in the data, e.g., cars always drive on the road, pedestrians do not walk into walls. Modeling logical formulas as computation graphs provides a natural way, in terms of language and computation, to incorporate contextual knowledge into the training process, thereby infusing desirable structure into the model which, as demonstrated through this example, can improve long-term prediction performance despite not having access to long-term data. We generate a dataset of signals s by adding noise to the tanh function (with random scaling and a random offset). With the first 10 time steps as inputs (i.e., trajectory history), we train a long short-term memory RNN to predict the next 10 time steps (i.e., future trajectory). This is visualized as the green and blue lines respectively in Figure 6. Suppose, based on contextual knowledge, we know *a priori* that the signal will eventually, beyond the next ten time steps, be in the interval $[0.4, 0.6]$. Specifically, the signal will satisfy $\phi = \lozenge \square_{[0,5]}(s > 0.4 \wedge s < 0.6)$. We can leverage knowledge of ϕ by rolling out the RNN model over a longer horizon, beyond what is provided in the training data, and apply the robustness loss on the rolled-out signal $s^{(i)}$ corresponding to the input $x^{(i)}$. We use (2) as the loss function where $\mathcal{L}_0$ is the

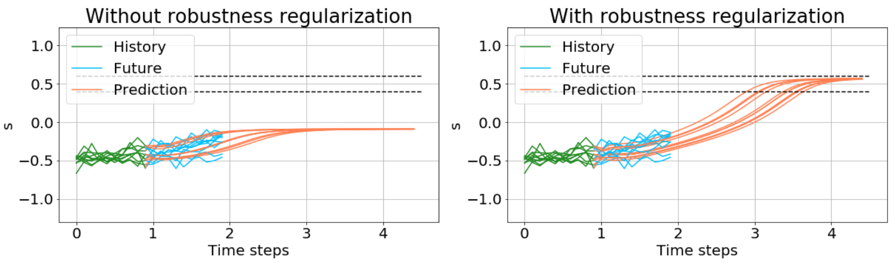


Fig. 6: Comparison of a sequence-to-sequence prediction model without (left) and with (right) robustness regularization. With robustness regularization, the model achieves desired long-term behavior $\phi = \Diamond \Box_{[0,5]} (s > 0.4 \wedge s < 0.6)$ despite having access to only short-term behaviors during training.

mean square error reconstruction loss over the first ten predicted time steps, and $\mathcal{L}_\rho = \sum_i \mathrm{ReLU}(-\rho(s^{(i)}, \phi))$ is the total (positive) amount of violations over the extended roll-out. With $\gamma = 0.1$, Figure 6 illustrates that with STL robustness regularization, the RNN model can successfully predict the next ten time steps and also achieve the desired long-term behavior despite being only trained on short-horizon data.

5 Future Work and Conclusions

We have described a technique called `stlcg` that transcribes STL robustness formulas as computation graphs. This enables the incorporation of logical specifications into problems in a way that is amenable to gradient computation for solution methods. As such, we are able to infuse logical structure in a diverse range of robotics applications such as motion planning, behavior clustering, and deep neural networks for model fitting and intent prediction. We highlight several directions for future work extending `stlcg` as presented in this paper. The first aims to extend the theory in order to enable optimization over parameters defining time intervals over which STL operators hold, and to also extend the language to express properties that cannot be expressed by the standard STL language. The second involves investigating how `stlcg` can help verify and improve the robustness of learning-based components in safety-critical settings governed by spatio-temporal rules, such as in autonomous driving and urban air-mobility contexts. The third is to investigate how to connect supervised structure introduced by logical specifications with unsupervised structure captured in the latent spaces of deep learning models. This may help provide interpretability via the lens of temporal logic into neural networks that are typically difficult to analyze.

References

1. Bartocci, E., Deshmukh, J., Donzé, A., Fainekos, G., Maler, O., Nickovic, D., Sankaranarayanan, S.: Specification-based monitoring of Cyber-Physical systems: A survey on theory, tools and applications. Lectures on Runtime Verification pp. 135–175 (2018)

2. Maler, O., Nickovic, D.: Monitoring temporal properties of continuous signals. In: Proc. Int. Symp. Formal Techniques in Real-Time and Fault-Tolerant Systems, Formal Modeling and Analysis of Timed Systems (2004)
3. Pant, Y.V., Abbas, H., Mangharam, R.: Smooth Operator: Control using the smooth robustness of temporal logic. In: IEEE Conf. Control Technology and Applications (2017)
4. Li, X., Vasile, C.I., Belta, C.: Reinforcement learning with temporal logic rewards. In: IEEE/RSJ Int. Conf. on Intelligent Robots & Systems (2017)
5. Raman, V., Donze, A., Maasoumy, M., Murray, R.M., Sangiovanni-Vincentelli, A., Seshia, S.A.: Model predictive control with signal temporal logic specifications. In: Proc. IEEE Conf. on Decision and Control (2014)
6. Mehr, N., Sadigh, D., Horowitz, R., Sastry, S., Seshia, S.A.: Stochastic predictive freeway ramp metering from signal temporal logic specifications. In: American Control Conference (2017)
7. Sadigh, D., Kapoor, A.: Safe control under uncertainty with probabilistic signal temporal logic. In: Robotics: Science and Systems (2016)
8. Chinchali, S.P., Livingston, S.C., Chen, M., Pavone, M.: Multi-objective optimal control for proactive decision-making with temporal logic models. Int. Journal of Robotics Research 38(12–13) (2019)
9. Yaghoubi, S., Fainekos, G.: Worst-case satisfaction of STL specifications using feedforward neural network controllers: A lagrange multipliers approach. ACM Transactions on Embedded Computing Systems 18(5s), 1–20 (2019)
10. Aksaray, D., Jones, A., Kong, Z., Schwager, M., Belta, C.: Q-learning for robust satisfaction of signal temporal logic specifications. In: Proc. IEEE Conf. on Decision and Control (2014)
11. Li, X., Ma, Y., Belta, C.: A policy search method for temporal logic specified reinforcement learning tasks. In: American Control Conference (2018)
12. Vazquez-Chanlatte, M., Deshmukh, J.V., Jin, X., Seshia, S.A.: Logical clustering and learning for time-series data. Proc. Int. Conf. Computer Aided Verification 10426, 305–325 (2017)
13. Asarin, E., Donzé, A., Maler, O., Nickovic, D.: Parametric identification of temporal properties. In: Int. Conf, on Runtime Verification (2012)
14. Liu, C., Arnon, T., Lazarus, C., Barrett, C., Kochenderfer, M.J.: Algorithms for verifying deep neural networks (2019), Available at https://arxiv.org/abs/1903.06758
15. Paszke, A., Gross, S., Chintala, S., Chanan, G., Yang, E., DeVito, Z., Lin, Z., Desmaison, A., Antiga, L., Lerer, A.: Automatic differentiation in PyTorch. In: Conf. on Neural Information Processing Systems - Autodiff Workshop (2017)
16. Fainekos, G., Sankaranarayanan, S., Ueda, K., Yazarel, H.: Verification of automotive control applications using S-TaLiRo. In: American Control Conference (2012)
17. Hochreiter, S., Schmidhuber, J.: Long short-term memory. Neural Computation (1997)
18. Mehdipour, N., Vasile, C.I., Belta, C.: Arithmetic-geometric mean robustness for control from signal temporal logic specifications. In: American Control Conference (2019)
19. Gers, F.A., Schmidhuber, J., Cummins, F.: Learning to forget: continual prediction with LSTM. In: Int. Conf. on Artificial Neural Networks (1999)
20. Schmerling, E., Leung, K., Vollprecht, W., Pavone, M.: Multimodal probabilistic model-based planning for human-robot interaction. In: Proc. IEEE Conf. on Robotics and Automation (2018)