STL2vec: Signal Temporal Logic Embeddings for Control Synthesis With Recurrent Neural Networks

Wataru Hashimoto, Kazumune Hashimoto, *Member, IEEE*, and Shigemasa Takai, *Member, IEEE*

***Abstract*—In this letter, a method for learning a recurrent neural network (RNN) controller that maximizes the robustness of signal temporal logic (STL) specifications is presented. In contrast to previous methods, we consider synthesizing the RNN controller for which the user is able to select an STL specification arbitrarily from multiple STL specifications. To obtain such a controller, we propose a novel notion called *STL2vec*, which represents a vector representation of the STL specifications and exhibits their similarities. The construction of the STL2vec is useful since it allows us to enhance the efficiency and performance of the controller. We validate our proposed method through the examples of the path planning problem.**

***Index Terms*—Signal temporal logic, neural network controller, optimal control.**

I. Introduction

Recently, formal methods have attracted much attention in control to address complex and temporal objectives such as periodic, sequential, or reactive tasks, going beyond the traditional control objectives like stability and tracking. Temporal logics such as linear temporal logic (LTL) [1], metric temporal logic (MTL) [2], and signal temporal logic (STL) [3] allow us to write down formal descriptions for these specifications.

In this letter, we focus on dealing with the STL specifications, which can specify the temporal properties of real-valued signals. One of the notable advantages of using the STL specifications in control is that we have access to the quantitative semantics, called *robustness*, which measures how much a signal satisfies the STL specification. The robustness has been used for control purposes in many existing works [4]–[10]. The authors of [4]–[6] proposed a scheme of encoding STL constraints or robustness of the STL specifications by mixed-integer linear constraints, and these are utilized to employ model predictive control (MPC). To alleviate the computational burden of mixed-integer programming, the works [7]–[10] proposed the concept of *smooth* robustness, allowing us to solve the maximization problem of the robustness based on gradient-based algorithms.

Manuscript received September 9, 2021; accepted January 10, 2022. Date of publication March 7, 2022; date of current version March 14, 2022. This letter was recommended for publication by Associate Editor Y. Zhao and Editor L. Pallottino upon evaluation of the reviewers' comments. This work was supported in part by JST CREST under Grant JPMJCR201, Japan, and in part by JSPS KAKENHI under Grant 21K14184. *(Corresponding author: Wataru Hashimoto.)*

The authors are with the Graduate School of Engineering, Osaka University, Suita 565-0871, Japan (e-mail: hashimoto@is.eei.eng.osaka-u.ac.jp; kazumune.hashimoto@z5.keio.jp; takai@eei.eng.osaka-u.ac.jp).

Digital Object Identifier 10.1109/LRA.2022.3155197

Aiming to further improve the scalability and meet real-time requirements, neural network (NN) based feedback controller was employed for STL tasks [11]–[15]. Although the learning procedure itself may be time-consuming, once this computation can be done offline, applying the resulting controller online becomes much faster than directly solving the optimization problem at each time step. In [11], the parameters of the feed-forward NN were trained to maximize the robustness of the STL specifications. Reinforcement learning algorithms such as deep Q-learning (DQN) and learning from demonstrations (Lfd) were used for learning the NN controller in [12] and [13] respectively. The works of [14], [15] used a recurrent neural network (RNN) [16] instead of a feed-forward NN, which is more suitable for control with STL since the satisfaction of an STL formula depends on a state trajectory (not just the current state). In [14], [15], RNN parameters were trained using imitation learning and model-based reinforcement, respectively.

However, the afore-cited previous works can deal with *only one* STL specification (i.e., the NN controller is trained for only one prescribed STL specification), and did not explicitly deal with multiple STL specifications. In particular, it may be more desirable and flexible in practice that the user is able to choose an STL specification freely from multiple STL specifications, such as the case where a surveillance task in a certain region (building, park, *etc.*) changes from day to day. A naive approach to achieve this would be to learn a set of NN controllers independently for all the candidate STL specifications, although this would be expensive in terms of memory consumption and computational resources if the number of the STL specifications is large (i.e., the number of the parameters to be learned could increase as the number of the candidate STL specifications increases). Hence, we here consider learning a *single* NN controller, in which not only the states of the system but also a chosen STL specification are regarded as inputs to the NN. We in particular focus on learning an RNN controller since as previously mentioned, it is suitable for dealing with history dependent nature of the STL specifications. The above formulation leads us to the following question to be answered, which has not been investigated in previous works of literature:

How should we encode the STL specifications by vectors, so that they are readable to the RNN?

Naively, we could assign any unique numbers or one-hot vectors to all the STL specifications and use them as the inputs to the RNN. Although these approaches are easy to implement, information about the similarities in the specifications cannot be encoded with these schemes. Thus, the underlying function to be learned can be unreasonably complex as the number of candidate STL specifications increases, which leads to increased computation time for training as well as a critical failure in control execution.

Motivated by the above issue, we propose a novel scheme for constructing a vector representation of STL specifications, called *STL2vec*, which captures similarities between the STL specifications. The vectors for the specifications will be trained based on Word2vec [17], [18] which is a technique widely known in natural language processing. Then, the parameters for the RNN are trained by taking the vectors obtained by STL2vec and state trajectory as the inputs (to the RNN) and defining a loss function by the average of negative robustness scores for all the candidate STL specifications.

The main contributions of this work are summarized as follows. First, we propose a method for obtaining vector representations of STL specifications that capture their similarities in terms of control policy. Although STL2vec is constructed based on the concept of word2vec, which is a technique that maps a word of the natural language onto the vector space, how to generate a dataset to learn appropriate vector representations for STL specifications in control synthesis is not a trivial problem. We address the details on how to construct dataset to train appropriate STL2vec in Section IV. Second, by using the vectors generated by STL2vec, we train a controller that can deal with multiple STL specifications with one RNN model. Naively, if one aims to train the NN controller to deal with multiple specifications by simply applying existing NN-based control synthesis methods such as [11]–[15], one needs to ready NN models as many as the number of the STL specifications, which leads to large memory consumption and computational resources. Moreover, the proposed method has the potential to accelerate the training as will be seen in the case study of Section V. Although there exist other methods that make NN controllers flexible such as [14] and [19], which can handle unknown obstacles by using control barrier function, these methods are restricted to only collision avoidance and thus cannot deal with multiple temporal logic specifications. The proposed method in this letter allows us to overcome such limitations; it allows the user to have flexibility of selecting an STL specification freely from multiple ones, and this will be achieved by constructing STL2vec.

II. Preliminaries

A. System Description and Notations

We consider a nonlinear discrete-time system of the form:

$$x_{t+1} = f(x_t, u_t),\ x_0 \in \mathcal{X}_0,\ u_t \in \mathcal{U}, \tag{1}$$

where $x_t \in \mathbb{R}^n$ and $u_t \in \mathcal{U}$ are the state and the control input at time $t \in \mathbb{Z}_{\geq 0}$, $\mathcal{X}_0 \subset \mathbb{R}^n$ is the set of initial states, $\mathcal{U} \subset \mathbb{R}^m$ is the set of control inputs, and $f : \mathbb{R}^n \times \mathcal{U} \to \mathbb{R}^n$ is a function capturing the dynamics of the system. We assume that the initial state x_0 is randomly chosen from $\mathcal{X}_0$ according to the probability distribution $p : \mathcal{X}_0 \to \mathbb{R}$, and $\mathcal{U}$ is given by $\mathcal{U} = \{u \in \mathbb{R}^m : u_{\min} \leq u \leq u_{\max}\}$ for given $u_{\max}, u_{\min} \in \mathbb{R}^m$ (the inequalities are interpreted element-wise). Given $x_0 \in \mathcal{X}_0$ and a sequence of control inputs $u_0, \ldots, u_{T-1}$, we can generate a unique sequence of states according to the dynamics (1), which we call a *trajectory:* $x_{0:T} = x_0, x_1, \ldots, x_T$.

B. Signal Temporal Logic

Signal temporal logic (STL) [3] is a logical formalism that can specify temporal properties of real-valued signals. The syntax of the STL formula is recursively defined as follows:

$$\begin{aligned}\phi ::= \top \mid \mu \mid \neg\phi \mid \phi_1 \wedge \phi_2 \mid \phi_1 \vee \phi_2 \mid \boldsymbol{F}_I\phi \mid \\ \boldsymbol{G}_I\phi \mid \phi_1 \boldsymbol{U}_I \phi_2\end{aligned} \tag{2}$$

where $\mu : \mathbb{R}^n \to \mathbb{B}$ is the predicate whose boolean truth value is determined by the sign of a function $h : \mathbb{R}^n \to \mathbb{R}$, $\top$, $\neg$, $\wedge$, and $\vee$ are Boolean *true*, *negation*, *and*, and *or* operators, respectively, and $\boldsymbol{F}_I$, $\boldsymbol{G}_I$, $\boldsymbol{U}_I$ are the temporal *eventually*, *always*, and *until* operators defined on a time interval $I = [a, b] = \{t \in \mathbb{Z}_{\geq 0} : a \leq t \leq b\}$ ($a, b \in \mathbb{Z}_{\geq 0}$). The definition of STL semantics is omitted in this letter and is provided in the extended version [24].

The notion of *robustness* of STL formulas provides *quantitative* semantics, and it measures how much the trajectory satisfies the STL formula. The robustness is sound in the sense that positive robustness value implies satisfaction of STL formula and negative robustness implies violation of STL formula. The robustness score of the STL formula ϕ is defined with respect to a trajectory $\mathbf{x} := x_{0:T}$ and a time t, which we denote by $\rho^{\phi}(\mathbf{x}, t)$, and is recursively defined as follows:

$$\begin{aligned}
\rho^{\mu}(\mathbf{x}, t) &= h(x_t) \\
\rho^{\neg\mu}(\mathbf{x}, t) &= -h(x_t) \\
\rho^{\phi_1 \wedge \phi_2}(\mathbf{x}, t) &= \min(\rho^{\phi_1}(\mathbf{x}, t), \rho^{\phi_2}(\mathbf{x}, t)) \\
\rho^{\phi_1 \vee \phi_2}(\mathbf{x}, t) &= \max(\rho^{\phi_1}(\mathbf{x}, t), \rho^{\phi_2}(\mathbf{x}, t)) \\
\rho^{\boldsymbol{F}_I\phi}(\mathbf{x}, t) &= \max_{t_1 \in t+I} \rho^{\phi}(\mathbf{x}, t) \\
\rho^{\boldsymbol{G}_I\phi}(\mathbf{x}, t) &= \min_{t_1 \in t+I} \rho^{\phi}(\mathbf{x}, t) \\
\rho^{\phi_1 \boldsymbol{U}_I \phi_2}(\mathbf{x}, t) &= \max_{t_1 \in t+I} \Big(\min(\rho^{\phi_2}(\mathbf{x}, t_1), \min_{t_2 \in [t, t_1]} \rho^{\phi_1}(\mathbf{x}, t_2))\Big)
\end{aligned}$$

where $t + I = \{t + k \in \mathbb{Z}_{\geq 0} : k \in I\}$. Note that the trajectory length T should be selected large enough to determine the robustness score (see, e.g., [4]). For simplicity, we denote $\rho^{\phi}(\mathbf{x})$ to abbreviate $\rho^{\phi}(\mathbf{x}, 0)$.

C. Word2vec

We summarize the basic concept of Word2vec [17] which is a key ingredient to introduce the proposed method discussed in Section IV and widely used in natural language processing tasks. The main objective of Word2vec is to group the vectors of similar or related words (for example, the words "man" and "boy" are mapped onto similar points in the vector space). As such, we can let the computer perform mathematical operations on words to detect their similarities. One of the most popular approaches for Word2vec is the skip-gram which has a shallow three-layer NN as shown in Fig. 1. The mapping from the input layer to the hidden layer is given by a matrix $W_{\text{in}} \in \mathbb{R}^{M \times N}$, where N is a

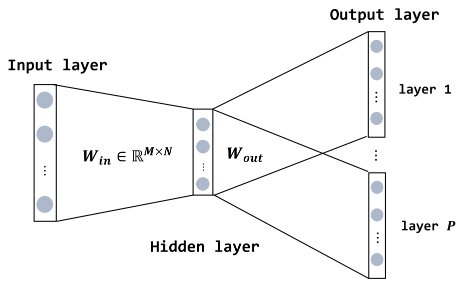


Fig. 1. Skip-gram model.

user-defined parameter that indicates the dimension of the word vector and M indicates a total number of words in the corpus (typically with $M > 10^6$). The mapping from the hidden layer to each output layer is given by another matrix $W_{\text{out}} \in \mathbb{R}^{N\times M}$. Although we consider P output layers as shown in Fig. 1 to learn the NN model with the dataset that we discuss in the following, we note here that the matrix W_{out} is shared for all the output layers. Motivated by the insight that the meaning of a word is established by the surrounding words, input and output data for training the NN model consist of all the pairs of a target word and corresponding P surrounding words in a large number of sentences, such as those in the news report (see, e.g. [17]). Here, each word is assigned by a unique M-dimensional one-hot vector so that each word is readable to the NN.

After the training, the vector representation of each word in the corpus is given through the projection of the weight matrix W_{in} (hence, we drop the matrix W_{out} in the skip-gram model). That is, for each word w in the corpus, the corresponding vector representation is given by $z_w = W_{\text{in}} e_w \in \mathbb{R}^N$, where $e_w \in \mathbb{R}^M$ denotes the one-hot vector of the word w. More details and some examples for Word2vec, please see the extended version [24].

III. Problem Statement

Let us now formulate a problem that we seek to solve throughout the letter. First, let $\Phi = \{\phi_1, \phi_2, \ldots, \phi_M\}$ denote a set of M *candidate* STL specifications. We assume that the STL specification can be freely chosen from the M candidate specifications by the user before the control execution. Once the STL specification is chosen by the user, say $\phi_i \in \Phi$, the system (1) is then controlled aiming to satisfy ϕ_i. We also assume for simplicity that the chosen STL specification is fixed during control execution (as detailed below).

In this letter, we aim to learn a feedback controller that maximizes the robustness of the STL specification. Note that the satisfaction and the robustness of an STL specification are defined over the trajectory of the system (1), and that the STL specification is freely chosen from Φ. Hence, we should design a control policy that depends not only on the past and present states, but also on the STL specification, i.e., we need to obtain a control policy of the form $u_t = \pi(x_{0:t}, \phi_i; W)$, where $x_{0:t} = x_0, x_1, \ldots, x_t$ is the trajectory of the system (1), ϕ_i is the STL specification that is chosen from Φ, and W denotes a set of parameters to be learned to characterize the control policy π.

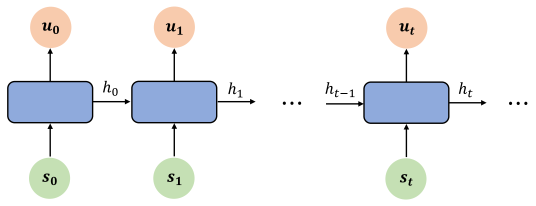


Fig. 2. The structure of RNN.

More formally, the problem considered in this letter is defined as follows:

Problem 1: Given the system (1), horizon length T, probability distribution of initial states $p : \mathcal{X}_0 \to \mathbb{R}$, and the candidate STL specifications $\Phi = \{\phi_1, \phi_2, \ldots, \phi_M\}$, find a set of parameters W that is the solution to the following problem:

$$\underset{W}{\text{maximize}} \ \frac{1}{M}\sum_{i=1}^{M} \mathbb{E}_{p(x_0^i)}\left[\rho^{\phi_i}\left(x_{0:T}^i\right)\right] \tag{3}$$

$$\text{s.t. } x_{t+1}^i = f\left(x_t^i, \pi\left(x_{0:t}^i, \phi_i; W\right)\right), \quad t = 0, 1, \ldots, T-1, \ i = 0, 1, \ldots, M. \tag{4}$$

where $\pi(\cdot, \cdot; W)$ denotes a control policy parameterized by W and $x_{0:t}^i = x_0^i, x_1^i, \ldots, x_t^i$ is the trajectory of (1) along with the policy $\pi(\cdot, \phi_i; W)$.

In Problem 1, we look for a set of the parameters of the control policy W that maximizes the sum of the expectation of the robustness score with respect to the distribution of initial states over all M candidate STL specifications. Since the control policy depends on the past and present states, or it is *history dependent*, in this letter we use a recurrent neural network (RNN) to learn the control policy π (see, [14], [15]). RNN is a type of a neural network that has a feedback architecture. A basic structure of the RNN is illustrated in Fig. 2. As shown in Fig. 2, RNN keeps processing the sequential information via internal hidden state h. The update rule of the hidden state and derivation of the output at time t are as follows:

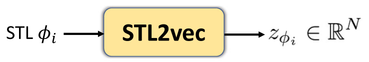
$$h_t = g_{W_1}(h_{t-1}, s_t), \ u_t = l_{W_2}(h_t), \tag{5}$$

where g_{W_1} and l_{W_2} are the functions parameterized by weights W_1 and W_2, respectively, and s_t denotes the input that is fed to the RNN. Hence, the set of the RNN parameters is given by $\{W_1, W_2\}$. As discussed in [11], we can restrict the output of the RNN (i.e., the control input u_t) within the lower bound u_{min} and the upper bound u_{max} by defining each element i $(i = 1, \ldots, m)$ of the function l_{W_2} as follows:

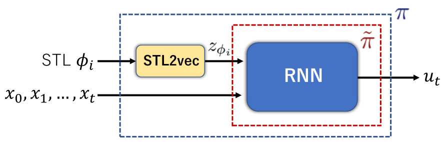
$$u_{t,i} = u_{\text{min},i} + \frac{u_{\text{max},i} - u_{\text{min},i}}{2}\left(\tanh\left([W_2 h_t]_i\right) + 1\right), \tag{6}$$

where the subscript i denote i-th element of the vector. Using (6), we can generate control inputs satisfying $u_t \in \mathcal{U}$. A detailed definition of s_t in (5) as well as a concrete procedure to learn W_1, W_2 are elaborated in Section IV.

Indeed, solving Problem 1 is not trivial and challenging in the following sense. Since the control policy depends on an STL specification ϕ_i, we should consider how to transform each STL specification $\phi_i \in \Phi$ into a corresponding *vector*, so that it can



(a) Step (i): train STL2vec, whose input is the STL specification $\phi_i \in \Phi$ and the output is the N-dimensional vector $z_{\phi_i} \in \mathbb{R}^N$.



(b) Step (ii): train RNN, whose inputs are a state trajectory $x_{0:t}$ and a vector representation of the STL specification z_{ϕ_i}, and the output is the control input u_t.

Fig. 3. Overview of the proposed approach.

be fed to the RNN as the inputs together with the state trajectory $x_{1:t}$. Intuitively, it may be desirable that we can provide similar inputs to the RNN if the two STL specifications are close to each other in terms of their control policies. For example, consider a simple case with $x \in \mathbb{R}$ and the STL candidate specifications

$$\phi_1 = \boldsymbol{F}_{[0,10]}(0 \leq x \leq 1),\ \phi_2 = \boldsymbol{F}_{[0,11]}(0 \leq x \leq 1), \quad (7)$$

$$\phi_3 = \boldsymbol{F}_{[0,10]}(10 \leq x \leq 11), \quad (8)$$

$$\phi_4 = \boldsymbol{F}_{[0,10]}(10 \leq x \leq 11) \vee \boldsymbol{F}_{[0,10]}(12 \leq x \leq 13). \quad (9)$$

Intuitively, control policies to satisfy ϕ_1 and ϕ_2 may be almost the same (as we aim to control the system to the same region with almost the same time intervals), while the control policies to satisfy ϕ_1 and ϕ_3 may be quite different (as we aim to control the system to different regions). In addition, if we could find a control policy to satisfy ϕ_3, this control policy also leads to the satisfaction of ϕ_4. Hence, it is convenient that we could provide a certain *mapping*, where ϕ_1 and ϕ_2 are mapped onto the vector points that are close to each other but far from those corresponding to ϕ_3 and ϕ_4. In addition, ϕ_3 and ϕ_4 are mapped onto the same vector points if a control policy to satisfy ϕ_3 could be found.

Motivated by the above intuition, in this letter we propose a novel scheme for constructing a *vector representation* of the STL specifications, which is referred to as *STL2vec*. The proposed approach is inspired by the notion of Word2vec [17] and elaborated in the next Section.

IV. Proposed Scheme

In this section, we describe the solution approach to Problem 1. The proposed approach consists of the two steps: (i) train a vector representation of the STL specifications, namely STL2vec, whose input is the STL specification $\phi_i \in \Phi$ and output is the N-dimensional vector; (ii) train the RNN, whose inputs are the state trajectory $x_{0:t}$ and the vector representation of the STL specification z_{ϕ_i}, and the output is the control input u_t. The illustration of the two steps are shown in Fig. 3. Since STL2vec will be trained based on the skip-gram, the parameter to be learned in Step (i) is W_{in} (recall in Section II-C that we neglect W_{out}). Also, recall that the parameters to be learned for RNN in Step (ii) is $\{W_1, W_2\}$ (see Section III). Hence, the overall parameters to be learned for the control policy π is $W = \{W_{\text{in}}, W_1, W_2\}$.

The proposed approach is beneficial in the following sense. By constructing STL2vec, we can obtain a vector representation that exhibits similarities between the STL specifications. This allows us to accelerate both efficiency and performance of the RNN controller in contrast to some naive approaches, such as integer or one-hot encoding i.e. simply assigning arbitrary numbers or one-hot vectors to the STL specifications (e.g., ϕ_1 is assigned by 1, ϕ_2 is assigned by 2, and so on) and use these numbers as the inputs to the RNN; for details, see an experimental result in Section V. Moreover, the proposed approach is more beneficial than the approach of learning RNN controllers one by one for each STL formula in terms of computational memory; for detailed analysis, see Section IV-C and Section V of this letter as well as Section V-C of the extended version [24]. The concrete procedures of the above two steps are given in the following subsections.

A. Training STL2vec

In this subsection, we provide a detailed procedure of Step (i) (train STL2vec). To train STL2vec, we use the skip-gram model as explained in Section II. First, we encode all the STL candidate specifications by the M-dimensional one-hot vectors, i.e. each STL specification $\phi_i \in \Phi$ is encoded by an M-dimensional vector whose i-th element is 1, and all other elements are 0 (ϕ_1 is encoded as $[1, 0, \dots, 0]^\mathsf{T}$ and ϕ_2 is encoded as $[0, 1, 0, \dots, 0]^\mathsf{T}$, and so on), so that they are readable to the NN. For simplicity, the one-hot vector of $\phi_i \in \Phi$ is denoted by $e_{\phi_i} \in \{0,1\}^M$.

A key question regarding the construction of STL2vec is how to learn the weight parameters $W_{\text{in}}, W_{\text{out}}$ for the skip gram model, or in other words, how to generate the training dataset to learn $W_{\text{in}}, W_{\text{out}}$. In this letter, we learn these parameters such that similar vector representations can be obtained if the two STL specifications are close to each other in terms of their control policies (for a detailed intuition for this, see Section III). To this end, we generate training dataset by comparing *robustness scores* among the STL specifications, aiming at measuring their closeness. More specifically, for each $\phi_i \in \Phi$, we randomly select an initial state $x_0 \in \mathcal{X}_0$ (following the probability distribution $p : \mathcal{X}_0 \to \mathbb{R}$), and solve the following maximization problem:

$$\underset{u_0,\dots,u_{T-1}}{\text{maximize}}\ \rho^{\phi_i}(x_{0:T})$$
$$\text{s.t.}\ \ x_{t+1} = f(x_t, u_t),\ u_t \in \mathcal{U},\ t = 0, 1, \dots, T-1. \quad (10)$$

Let $u_0^*, u_1^*, \dots, u_{T-1}^* \in \mathcal{U}$ denote the optimal control inputs as the solution to (10), and $x_{0:T}^* = x_0^*, x_1^*, \dots, x_T^*$ the corresponding trajectory of the system (1). Thus, the corresponding (maximized) robustness score is $\rho^{\phi_i}(x_{0:T}^*)$. Then, we compute the robustness scores of the obtained trajectory $x_{0:T}^*$ with respect to *all the other* STL specifications, i.e.,

$$\rho^{\phi_j}(x_{0:T}^*),\ \text{for all}\ \phi_j \in \Phi\ \text{with}\ i \neq j. \quad (11)$$

Algorithm 1: Learning $W_{\rm in}, W_{\rm out}$ in the skip-gram.

```
Input  : $\Phi = \{\phi_1, \ldots, \phi_M\}$ (candidate specifications); $x_0$ (initial state); $P$ (number of output layers); $N$ (dimension of vector representation); $N_{\rm ite}$ (number of iterations)
Output: $W_{\rm in}, W_{\rm out}$ (weight parameters for the skip-gram)
1  $\mathcal{D} \leftarrow \varnothing$;
2  for each $\phi_i \in \Phi$ do
3      for $\ell = 1 : N_{\rm ite}$ do
4          Select $x_0 \in \mathcal{X}_0$ according to the probability distribution $p : \mathcal{X}_0 \to \mathbb{R}$;
5          Given $\phi_i \in \Phi$, $x_0 \in \mathcal{X}_0$, solve (10) to obtain the optimal trajectory $x^*_{0:T} = x^*_0, \ldots, x^*_T$;
6          Compute the robustness scores of $x^*_{0:T}$ with respect to the other STL specifications (11);
7          Sort (11) in order of their closeness to $\rho^{\phi_i}(x^*_{0:T})$, and pick up the first $P$ STL specifications: $\phi^i_{k_1}, \phi^i_{k_2}, \ldots, \phi^i_{k_P}$;
8          Let $\mathcal{D}_{\rm temp}$ be given by (12). Then, update the dataset as $\mathcal{D} \leftarrow \mathcal{D} \cup \mathcal{D}_{\rm temp}$;
9      end
10 end
11 Based on the training data set $\mathcal{D}$ and the cross entropy loss, train $W_{\rm in}, W_{\rm out}$ via back propagation.
```

Then, we sort the robustness scores of (11) in order of their closeness to $\rho^{\phi_i}(x^*_{0:T})$ (i.e., sort by evaluating $|\rho^{\phi_i}(x^*_{0:T}) - \rho^{\phi_j}(x^*_{0:T})|$, $i \neq j$). We denote the ordered specifications by $\phi^i_{k_1}, \phi^i_{k_2}, \ldots, \phi^i_{k_{M-1}}$, where $\phi^i_{k_j} \in \Phi$ denotes the specification which has j-th closest robustness value to $\rho^{\phi_i}(x^*_{0:T})$.

Then, we pick up the first P STL specifications, i.e., $\phi^i_{k_1}, \phi^i_{k_2}, \ldots, \phi^i_{k_P}$ (recall that P is the number of output layers in the skip-gram model), and set the input and output training data as $\{\phi_i, (\phi^i_{k_1}, \phi^i_{k_2}, \ldots, \phi^i_{k_P})\}$, where ϕ_i represents the input data and $(\phi^i_{k_1}, \phi^i_{k_2}, \ldots, \phi^i_{k_P})$ represents the output data. This data is then encoded by the one-hot vectors, regarded as the training data to learn the skip-gram:

$$\left\{e_{\phi_i}, \left(e_{\phi^i_{k_1}}, e_{\phi^i_{k_2}}, \ldots, e_{\phi^i_{k_P}}\right)\right\}. \tag{12}$$

The data in (12) is then added to the dataset.

For each $\phi_i \in \Phi$, we repeat the above process $N_{\rm ite}$ times so as to obtain a collection of the training data.

The proposed approach presented above is summarized in Algorithm IV-A. Then, the weight parameters $W_{\rm in}, W_{\rm out}$ are learned by minimizing the cross entropy loss via back-propagation.

Finally, similarly to Word2vec, the vector of each specification is given through the projection of $W_{\rm in}$, i.e., the vector representation of $\phi_i \in \Phi$ is given by $z_{\phi_i} = W_{\rm in} e_{\phi_i}$.

Remark 1: Since robustness can be nested with max/min functions, the problem (10) can be non-differentiable. To deal with this, we use *smooth approximation* of the min/max operators (see the extended version [24]).

B. Training RNN

Here, we describe a detailed procedure of Step (ii). Recall that in Step (ii), we aim to train RNN, whose inputs are the trajectory $x_{0:t}$ and the vector representation of ϕ_i (i.e., z_{ϕ_i}), and the output is the control input u_t. Further, remember that the concrete unfolded structure of the RNN is shown in Fig. 2, and the update rules are given by (5)–(6). Here, the input variable s_t is given by the concatenation of the state x_t and the vector representation of the chosen STL specification, i.e., $s_t = [x_t^\top, z_{\phi_i}^\top]^\top$, where $z_{\phi_i} \in \mathbb{R}^N$ is the vector representation of the chosen STL specification ϕ_i. Now, let $\tilde{\pi}(x_{0:t}, z_{\phi_i}; W_1, W_2)$ denote the control policy (or a mapping) for the RNN, where W_1, W_2 are the RNN parameters to be learned (for the illustration, see Fig. 3(b)). Since the parameters for STL2vec $W_{\rm in}$ is fixed, we here focus on learning the RNN parameters W_1, W_2 to characterize $\tilde{\pi}$.

To numerically solve the Problem 1, we keep repeating the following procedure until a certain number of epochs $N_{\rm epo}$ is reached: First, in each epoch, we construct *mini-batch* of the specifications by randomly rearranging the order of the candidate specifications and splitting them up according to the prespecified batch size N_b, i.e., we construct the batches $\mathcal{B}_1 = \{\phi^1_1, \ldots, \phi^1_{N_b}\}, \mathcal{B}_2 = \{\phi^2_1, \ldots, \phi^2_{N_b}\}, \dot{s}, \mathcal{B}_K = \{\phi^K_1, \ldots, \phi^K_{N_b}\}$, where $K = \frac{M}{N_b}$ (assuming that M can be divided by N_b) [1]. Then, we iterate the following procedures for all the batches $\mathcal{B}_p$ $(p = 1, \ldots, K)$ to update the parameters W_1 and W_2.

1) Forward Computation: For all the pairs of the initial state x^j_0 $(j = 1, \ldots, L)$ which are randomly sampled from the initial region $\mathcal{X}_0$ and the vectors of the specifications $z_{\phi^p_i}$ $(i = 1, \ldots, N_b)$ in a batch $\mathcal{B}_p$, we generate the trajectories $x^{i,j}_{0:T}$ by iteratively applying $x^{i,j}_{t+1} = f(x^{i,j}_t, \tilde{\pi}(x^{i,j}_{0:t}, z_{\phi^p_i}; W_1, W_2))$ for fixed W_1 and W_2 from the initial state $x^{i,j}_0 = x^j_0$. Then, by using all the $N_b L$ trajectories obtained above, we compute the following loss:

$$-\frac{1}{N_b L}\sum_{i=1}^{N_b}\sum_{j=1}^{L}\rho^{\phi^p_i}\left(x^{i,j}_{0:T}\right), \tag{13}$$

which is an approximation of the expectation (3) in Problem 1 with the specifications in a batch (note that we take the negative of the robustness to define a loss).

2) Backward Computation: After the forward computation, we compute the gradients for all the parameters via backpropagation through time (BPTT) [16]. This computation can be easily implemented by combining the auto-differentiation tools designed for NNs like PyTorch [22] and STLCG [20] which is a newly developed python toolbox for computing the STL robustness using computation graph.

3) Weight Update: Lastly, we update all the parameters in the RNN controller by using the weights obtained above. In this letter, we use Adam optimizer [21].

C. Comparison Regarding Computation Memory

The proposed approach has the potential to require much less computational memory than learning the RNN controllers

[1] If M is not divisible by N_b, we construct minibatch with $K = \lfloor \frac{M}{N_b} \rfloor$ ($\lfloor\cdot\rfloor$ denotes a floor function) and append the remaining specifications in the last batch. For example, if $\Phi = \{\phi_1, \phi_2, \ldots, \phi_5\}$, and $N_b = 2$, we construct minibatch e.g., $\mathcal{B}_1 = \{\phi_1, \phi_3\}$, $\mathcal{B}_2 = \{\phi_2, \phi_4, \phi_5\}$.

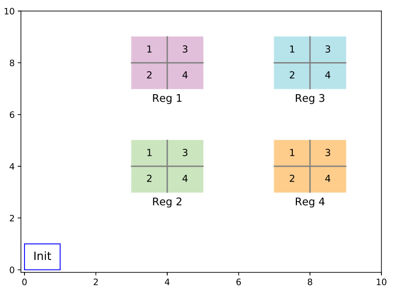


Fig. 4. Regions and sub-regions in a 2D space.

for all the specifications one-by-one due to the following reason. In the proposed method, the total number of the parameters to be learned is given by $N_{\rm in}+N_{R1}+N_{R2}$, where $N_{\rm in}$, N_{R1}, and N_{R2} are the number of elements in $W_{\rm in}$, W_1, and W_2. For instance, when we use the Long-Short-Term-Memory (LSTM) as the RNN model, N_{R1} and N_{R2} are given by $N_{R1}=4N_hN_{\rm lstm}(n+N)$ and $N_{R2}=4N_hN_{\rm lstm}m$, where N_h is the dimension of the hidden state of RNN and $N_{\rm lstm}$ is the number of LSTM layers. The number of parameters for STL embedding $N_{\rm in}$ is given by $N_{\rm in}=MN$. On the other hand, the required number of parameters when we train the controller one-by-one for each candidate specifications is given by $M(N'_{R1}+N_{R2})$, where $N'_{R1}=4N_hN_{\rm lstm}n$. Although the number of the parameters is proportional to the number of STL specifications M for both the approaches, we typically have $N \ll N'_{R1}+N'_{R2}$, and thus the number of the required parameters by the proposed approach is smaller than the one-by-one approach. For more details, please see arxiv version [24].

V. Case Study

We show the efficacy of the proposed method through an example of a path planning problem of a vehicle in 2D space. The reader is referred to the extended manuscript [24] for more detailed analysis and experimental evaluations. We used PyTorch package [22] for the implementation of RNN, IPOPT in CasADi [23] to solve optimization (10), and STLCG [20] for the computation regarding STL robustness. We used Windows 10 with a 2.80 GHz Core i7 CPU and 32 GB of RAM for all the experiments given in this section. We consider the following nonlinear, discrete-time nonholonomic system: $q_{x,t+1}=q_{x,t}+v_t\sin\theta_t$, $q_{y,t+1}=q_{y,t}+v_t\cos\theta_t$, $\theta_{t+1}=\theta_t+\omega_t$, where $q_{x,t}$, $q_{y,t}$ represent position of the vehicle, θ_t is the heading angle, v_t and ω_t are velocity and angular velocity, respectively. The state x_t and control input u_t are defined as $x_t=[q_{x,t},q_{y,t},\theta_t]^\top$ and $u_t=[v_t,\omega_t]^\top$, respectively. As illustrated in Fig. 4, we consider 4 regions in a 2D space: Reg $1=[3,5]\times[7,9]$, Reg $2=[3,5]\times[3,5]$, Reg $3=[7,9]\times[3,5]$, Reg $4=[7,9]\times[7,9]$, and the set of initial states $\mathcal{X}_0=[0,0.7]\times[0,0.7]$ (blue-edged color region). In addition, we consider 4 sub-regions in each region, which are labeled by indices as shown in Fig. 4. We denote j-th sub-region $(j=1,2,3,4)$ in region i $(i=1,2,3,4)$ as Reg (i,j).

The STL candidate specifications are shown in Table I. Here, the specifications are considered for all $i,i_1,i_2,i_3\in\{1,\dots,4\}$

TABLE I
STL Candidate Specifications

Specifications
(a) $\boldsymbol{F}_{[0,20]}$ Reg (i,j)
(b) $\boldsymbol{F}_{[0,20]}$ Reg $(i_1,j_1)\vee\boldsymbol{F}_{[0,20]}$ Reg (i_2,j_2)
(c) $\boldsymbol{F}_{[0,10]}$ Reg $(i_1,j_1)\wedge\boldsymbol{F}_{[11,20]}$ Reg (i_3,j_2)
(d) $\boldsymbol{F}_{[0,15]}\boldsymbol{G}_{[0,5]}$ Reg (i,j)
(e) $\boldsymbol{F}_{[0,15]}\boldsymbol{G}_{[0,5]}$ Reg $(i_1,j_1)\vee\boldsymbol{F}_{[0,15]}\boldsymbol{G}_{[0,5]}$ Reg (i_2,j_2)
(f) $\boldsymbol{F}_{[0,20]}$ Reg $(4,4)\wedge(\neg$Reg $(4,4)$ $\boldsymbol{U}_{[0,20]}$ Reg $(2,3))$

TABLE II
The Time Required to Obtain STL2vec (In Sec)

Procedure	Run-time
Dataset generation (STL2vec)	1452
Training STL2vec	87

$(i_1>i_2, i_3\neq 1, i_1\neq i_3)$ and $j,j_1,j_2,j_3\in\{1,\dots,4\}$. The total number of the STL candidate specifications is given by 369.

A. Training STL2vec

Now we train STL2vec based on the proposed approach presented in Section IV-A. The parameters for the skip-gram are $N=20$ and $P=2$. $N_{\rm ite}$ in Algorithm IV-A and the number of epochs for the training of skip-gram model are set to 1 and 100, respectively. After the training, we evaluate similarities between all the 2 different vector representations z_{ϕ_i},z_{ϕ_j} $(i\neq j)$ by the cosine similarity, which is defined as $(z_{\phi_i}^\top z_{\phi_j})/(\|z_{\phi_i}\|\|z_{\phi_j}\|)$ and it takes the maximum value of 1 if z_{ϕ_i} has the same orientation as z_{ϕ_j}. The time required to train STL2vec is summarized in Table II. In Table III, we illustrate STL specifications which have the largest to fourth-largest cosine similarities for some example specifications.

We summarize some characteristics that we have observed in the obtained embeddings in the followings: (I) Each specification in (a) of Table I is typically embedded close to the corresponding specification in (d). Moreover, each specification in (b) and (e) is embedded close to either $\boldsymbol{F}_{[0,20]}$ Reg (i_1,j_1) (and $\boldsymbol{F}_{[0,15]}\boldsymbol{G}_{[0,5]}$Reg (i_1,j_1)) or $\boldsymbol{F}_{[0,20]}$ Reg (i_2,j_2) (and $\boldsymbol{F}_{[0,15]}\boldsymbol{G}_{[0,5]}$Reg (i_2,j_2)). Typically, (b) and (e) are embedded close to the specification regarding the region which is closer to init region. For example, as we can partially see in Ex 1 of Table III, we have observed that almost all the specifications $\boldsymbol{F}_{[0,20]}$ Reg $(2,2)\vee\boldsymbol{F}_{[0,20]}$ Reg (i,j) and $\boldsymbol{F}_{[0,15]}$ $\boldsymbol{G}_{[0,5]}$ Reg $(2,2)\vee\boldsymbol{F}_{[0,15]}$ $\boldsymbol{G}_{[0,5]}$ Reg (i,j) $(i\in\{1,3,4\},\ j\in\{1,2,3,4\})$ are embedded close to the specification $\boldsymbol{F}_{[0,20]}$ Reg $(2,2)$. This is intuitive and desirable result when we train the controller since all of these specifications are satisfied by entering whether Reg (i_1,j_1) or Reg (i_2,j_2) and stay there more than 5 steps, which is possible actions for all of regions in this example. (II) Each specification in (c) of Table I is basically embedded close to the specifications which are satisfied by similar trajectory. Specifically, the properties same as the following examples are observed for all the specifications in (c): (i) specifications $\boldsymbol{F}_{[0,10]}$ Reg $(1,3)\wedge\boldsymbol{F}_{[11,20]}$ Reg $(3,j)$ $(j\in\{1,2,3,4\})$ are embedded close to each other (see Ex 2 of Table III); (ii) as we can see in Ex 3 of Table III, the specifications $\boldsymbol{F}_{[0,10]}$ Reg $(2,2)\wedge\boldsymbol{F}_{[11,20]}$ Reg $(3,1)$ and $\boldsymbol{F}_{[11,20]}$ Reg $(3,1)$

TABLE III
COSINE SIMILARITIES OF VECTOR REPRESENTATIONS OF THE STL CANDIDATE SPECIFICATIONS

Ex 1	$\boldsymbol{F}_{[0,20]}$ Reg $(2,2)$	sim	Ex 2	$\boldsymbol{F}_{0,10]}$ Reg $(1,3) \wedge \boldsymbol{F}_{[11,20]}$ Reg $(3,1)$	sim
1	$\boldsymbol{F}_{[0,20]}$ Reg $(3,3) \vee \boldsymbol{F}_{[0,20]}$ Reg $(2,2)$	0.99	1	$\boldsymbol{F}_{[0,10]}$ Reg $(1,3) \wedge \boldsymbol{F}_{[11,20]}$ Reg $(3,4)$	0.99
2	$\boldsymbol{F}_{[0,20]}$ Reg $(2,2) \vee \boldsymbol{F}_{[0,20]}$ Reg $(1,1)$	0.99	2	$\boldsymbol{F}_{[0,10]}$ Reg $(1,3) \wedge \boldsymbol{F}_{[11,20]}$ Reg $(3,3)$	0.99
3	$\boldsymbol{F}_{[0,15]}\boldsymbol{G}_{[0,5]}$ Reg $(4,4) \vee \boldsymbol{F}_{[0,15]}\boldsymbol{G}_{[0,5]}$ Reg $(2,2)$	0.99	3	$\boldsymbol{F}_{[0,10]}$ Reg $(1,3) \wedge \boldsymbol{F}_{[11,20]}$ Reg $(3,2)$	0.95
4	$\boldsymbol{F}_{[0,15]}\boldsymbol{G}_{[0,5]}$ Reg $(2,2)$	0.99	4	$\boldsymbol{F}_{[0,15]}\boldsymbol{G}_{[0,5]}$ Reg $(3,3) \vee \boldsymbol{F}_{[0,15]}\boldsymbol{G}_{[0,5]}$ Reg $(1,3)$	0.76
Ex 3	$\boldsymbol{F}_{[0,10]}$ Reg $(2,2) \wedge \boldsymbol{F}_{[11,20]}$ Reg $(3,1)$	sim	Ex 4	$\boldsymbol{F}_{[0,20]}$ Reg $(4,4) \wedge \left(\neg\text{Reg }(4,4)\ \boldsymbol{U}_{[0,20]}\text{ Reg }(2,3)\right)$	sim
1	$\boldsymbol{F}_{[0,20]}$ Reg $(3,1)$	0.96	1	$\boldsymbol{F}_{[0,10]}$ Reg $(2,3) \wedge \boldsymbol{F}_{[11,20]}$ Reg $(4,4)$	0.81
2	$\boldsymbol{F}_{[0,20]}$ Reg $(3,1) \vee \boldsymbol{F}_{[0,20]}$ Reg $(1,1)$	0.95	2	$\boldsymbol{F}_{[0,10]}$ Reg $(4,4) \wedge \boldsymbol{F}_{[11,20]}$ Reg $(3,3)$	0.73
3	$\boldsymbol{F}_{[0,20]}$ Reg $(4,4) \vee \boldsymbol{F}_{[0,20]}$ Reg $(3,1)$	0.95	3	$\boldsymbol{F}_{[0,10]}$ Reg $(4,4) \wedge \boldsymbol{F}_{[11,20]}$ Reg $(3,1)$	0.71
4	$\boldsymbol{F}_{[0,15]}\boldsymbol{G}_{[0,5]}$ Reg $(4,4) \vee \boldsymbol{F}_{[0,15]}\boldsymbol{G}_{[0,5]}$ Reg $(3,1)$	0.93	4	$\boldsymbol{F}_{[0,10]}$ Reg $(4,4) \wedge \boldsymbol{F}_{[11,20]}$ Reg $(3,4)$	0.71

TABLE IV
SOME ALTERNATIVE APPROACHES

Approaches
(A1) Integer encoding scheme
(A2) One-hot encoding scheme
(A3) Training controllers one-by-one

are mapped onto similar vectors (these specifications are satisfied with the same trajectory since Reg$(2,2)$ is on the way from the starting point to Reg$(3,1)$).

B. Training RNN

Next, we evaluate the control performance of the proposed method. In this experiment, we train the parameters of RNN model for the specifications (a), (b), (c), (f) and $\boldsymbol{F}_{[0,15]}\boldsymbol{G}_{[0,5]}$ Reg $(1,1) \vee \boldsymbol{F}_{[0,15]}\boldsymbol{G}_{[0,5]}$ Reg $(3,2)$ in (e) in Table I. As for the specifications in (c), we only consider the specifications with $(i_1,i_3) = (1,3),(2,1),(2,3),(2,4),(4,3)$. Thus the total number of specifications is 194. We retrain the STL embeddings (20 dimensional) with these specifications and use them as the input to the RNN. Same as the previous work [15], all of the RNN models used in this example consists of 2 LSTM layers with 32-dimensional hidden states. As summarized in Table IV, we consider the following 3 alternative approaches for comparison with the proposed method: (A1) Integer encoding scheme: we incrementally assign an integer number for each STL specification (for example, $\boldsymbol{F}_{[0,20]}$ Reg $(1,1)$ is assigned by 1, $\boldsymbol{F}_{[0,20]}$ Reg $(1,2)$ is assigned by 2, *etc*), and use it as the input to the RNN (instead of the vectors generated by STL2vec) (A2) One-hot encoding scheme: we generate and assign 194 dimensional one-hot vectors for all the 194 specifications (for example, $\boldsymbol{F}_{[0,20]}$ Reg $(1,1)$ is assigned by $[1,0,\ldots,0]$, $\boldsymbol{F}_{[0,20]}$ Reg $(1,2)$ is assigned by $[0,1,\ldots,0]$, *etc*) (A3) Training one-by-one: we ready 194 RNN models and train each specification one-by-one. When we train the controllers for the approaches (A1) and (A2), we used the same training procedure discussed in Section IV-B and used integer numbers or one-hot vectors instead of the vectors obtained by STL2vec. In both the proposed approach and (A1), (A2), we set the batch size and the number of initial states sampled for training in each iteration (N_b and L in Section IV-B)) to 8 and 3, respectively. In (A3), for each epoch we generate initial states x_0^j ($j = 1,\ldots,L = 3$) randomly from $\mathcal{X}_0$ and update the RNN parameters assigned for each STL specification $\phi_i \in \Phi$ ($i = 1,\ldots,M$) one by one via forward/backward computation

TABLE V
RUNTIME NEEDED TO REACH SEVERAL ROBUSTNESS VALUES (IN SEC)

approach \average of robustness	0.1	0.15	0.2	0.22
Proposed method	3494	4054	6722	14668
Learning one-by-one	5686	7663	10336	12168
One-hot encoding	27238	33863	–	–
Integer encoding	–	–	–	–

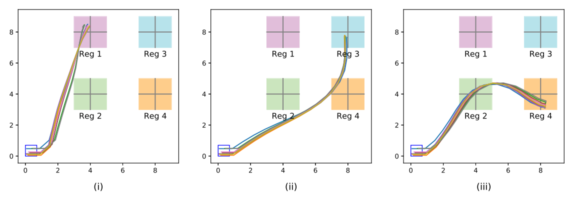
TABLE VI
TOTAL NUMBER OF PARAMETERS IN EACH METHOD

	Number of parameters
Proposed method	10280
approach (A1)	1280
approach (A2)	50432
approach (A3)	248320

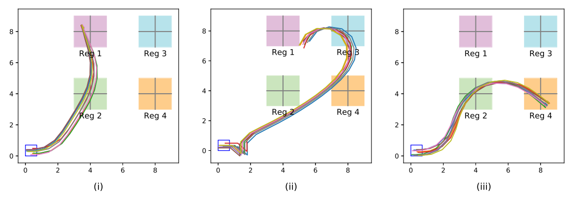
(similarly to the procedure of Section V-B) with the following loss: $-\frac{1}{L}\sum_{j=1}^{L}\rho^{\phi_i}(x_{0:T}^j)$.

Table V summarizes the actual time (in sec) required to reach the robustness values 0.1, 0.15, 0.2, 0.22 for all the proposed and alternative methods (the symbol "-" indicates that the corresponding average of robustness has not been achieved). The average of robustness is evaluated by testing the controller with 30 initial states randomly sampled from $\mathcal{X}_0$ and the time for approach (A3) is defined by the total sum of the times required to update the parameters of all the RNN models. Furthermore, the resulting few trajectories obtained by applying the RNN controller trained by the proposed method and approach (A3) are plotted in Fig. 5 (both controllers were trained 1100 epochs and the trajectories are plotted for 10 initial states newly sampled from $\mathcal{X}_0$).

As can be seen from Table V, the average of robustness for the proposed method reaches 0.1, 0.15, and 0.2 faster than the other approaches. Moreover, as mentioned in Section V-C of [24], we can largely save the memory consumption compared with (A3). In Table VI, we summarize the total number of parameters required for each approach in this example. However, the average of robustness value of the approach (A3) reaches 0.22 faster than the proposed approach. Within 1100 epochs, the maximum averaged robustness values of the proposed method and approach (A3) were 0.233 and 0.235, respectively. The reason why the averaged robustness of the proposed method is overtaken by that of the approach (A3) may be because of the effect of the other specifications, i.e., since only one RNN controller is trained for many specifications in the proposed method,



(a) Trajectories for the proposed method.



(b) Trajectories for the approach (A3)

Fig. 5. The testing result for 10 initial states and the 3 specifications (i) $F_{[0,15]}G_{[0,5]}\text{Reg}\ (1,1) \vee F_{[0,15]}G_{[0,5]}\text{Reg}\ (3,2)$, (ii) $F_{[0,10]}\text{Reg}\ (4,1) \wedge F_{[10,20]}\text{Reg}\ (3,2)$, (iii) $\boldsymbol{F}_{[0,20]}\ \text{Reg}\ (4,4) \wedge (\neg\text{Reg}\ (4,4)\ \boldsymbol{U}_{[0,20]}\ \text{Reg}\ (2,3))$ are plotted. The results are shown for both (a) proposed scheme and (b) approach (A3).

the control performance for a specification may be affected by the other specifications depending on the obtained embedding. Such effect is observed in (b) of Fig. 5. The resulting trajectories are converged to the ones that are relatively far from the middle of $\text{Reg}(3,2)$, which leads to the low robustness although the specification itself is satisfied. To remove such behavior, further investigation for obtaining more superior embedding will be one of our future works.

VI. Conclusion and Future Work

We proposed a method for mapping STL specifications onto the vector space (STL2vec) based on the word2vec technique. To obtain the STL embeddings that capture the similarities in specifications in terms of control policy, we have provided a method for constructing the dataset by solving the robustness maximization problem for all the candidate specifications. Then, we trained the RNN controller whose inputs are the state trajectory and a vector generated by STL2vec to deal with multiple STL specifications with a single RNN model. The example shown in the simulation section shows the efficacy of the proposed method in terms of memory consumption and the time required for the training.

In this letter, it is assumed that the chosen STL specification is fixed during control execution and not allowed to change during the execution. Hence, future work should involve investigating the case where the STL specification is changed during control execution.

References

[1] A. Pnueli, "The temporal logic of programs," in *Proc. 18th Annu. Symp. Foundations Comput. Sci.*, 1977, pp. 46–57.
[2] R. Koymans, "Specifying real-time properties with metric temporal logic," *Real-Time Syst.*, vol. 2, no. 4, pp. 255–299, 1990.
[3] O. Maler and D. Nickovic, "Monitoring temporal properties of continuous signals," in *Proc. Formal Techn., Modelling Anal. Timed Fault-Tolerant Syst.*, 2004, pp. 152–166.
[4] V. Raman, A. Donze, M. Maasoumy, R. M. Murray, A. Sangiovanni-Vincentelli, and S. A. Seshia, "Model predictive control with signal temporal logic specifications,"in *Proc. IEEE Conf. Decis. Control*, 2014, pp. 81–87.
[5] S. Sadraddini and C. Belta, "Robust temporal logic model predictive control," in *Proc. IEEE Annu. Allerton Conf. Commun., Control, Comput.*, 2015, pp. 772–779.
[6] N. Mehr, D. Sadigh, R. Horowitz, S. S. Sastry, and S. A. Seshia, "Stochastic predictive freeway ramp metering from signal temporal logic specifications," in *Proc. IEEE Amer. Control Conf.*, 2017, pp. 4884–4889.
[7] Y. V. Pant, H. Abbas, and R. Mangharam, "Smooth operator: "Control using the smooth robustness of temporal logic," in *Proc. IEEE Conf. Control Technol. Appl.*, 2017, pp. 1235–1240.
[8] X. Li, Y. Ma, and C. Belta, "A policy search method for temporal logic specified reinforcement learning tasks," in *Proc. IEEE Amer. Control Conf.*, 2018, pp. 240–245.
[9] N. Mehdipour, C. Vasile and C. Belta, "Arithmetic-geometric mean robustness for control from signal temporal logic specifications," in *Proc. IEEE Amer. Control Conf*, 2019, pp. 1690–1695.
[10] I. Haghighi, N. Mehdipour, E. Bartocci, and C. Belta, "Control from signal temporal logic specifications with smooth cumulative quantitative semantics," in *Proc. IEEE Conf. Decis. Control*, 2019, pp. 4361–4366.
[11] S. Yaghoubi and G. Fainekos, "Worst-case satisfaction of stl specifications using feedforward neural network controllers: A lagrange multipliers approach," in *Proc. Inf. Theory Appl. Workshop*, 2020, pp. 1–20.
[12] A. Balakrishnan and J. V. Deshmukh, "Structured reward shaping using signal temporal logic specifications," in *Proc. IEEE/RSJ Int. Conf. Intell. Robots Syst.*, 2019, pp. 3481–3486.
[13] A. Gopinath Puranic, J. V. Deshmukh and S. Nikolaidis, "Learning from demonstrations using signal temporal logic in stochastic and continuous domains," *IEEE Robot. Automat. Lett.*, vol. 6, no. 4, pp. 6250–6257, Oct. 2021.
[14] W. Liu, N. Mehdipour, and C. Belta, "Recurrent neural network controllers for signal temporal logic specifications subject to safety constraints," *IEEE Control. Syst. Lett*, vol. 6, pp. 91–96, 2022, doi: 10.1109/LCSYS.2021.3049917.
[15] W. Liu and C. Belta, "Model-based safe policy search from signal temporal logic specifications using recurrent neural networks," 2021, *arXiv:2103.15938*.
[16] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural Comput.*, vol. 9, no. 8, pp. 1735–1780, 1997.
[17] T. Mikolov, I. Sutskever, K. Chen, G. S. Corrado, and J. Dean, "Distributed representations of words and phrases and their compositionality," in *Proc. Adv. Neural Inf. Process. Syst.*, 2013, pp. 3111–3119.
[18] X. Rong, "word2vec parameter learning explained," 2016, *arXiv:1411.2738*.
[19] L. Xiao, Z. Serlin, G. Yang, and C. Belta, "A formal methods approach to interpretable reinforcement learning for robotic planning." *Sci. Robot.*, vol 4, no. 37 2019, Art. no. eaay6276.
[20] K. Leung, N. Arechiga, and M. Pavone, "Back-propagation through signal temporal logic specifications: Infusing logical structure into gradient-based methods," *Workshop Algorithmic Foundations Robot.*, 2020.
[21] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," in *Proc. Int. Conf. Mach. Learn.*, 2015.
[22] A. Paszke *et al.*, "Automatic differentiation in pytorch," in *Proc. Neural Inf. Process. Syst.*, 2017.
[23] https://web.casadi.org
[24] W. Hashimoto, K. Hashimoto, and S. Takai, "STL2vec: Signal temporal logic embeddings for control synthesis with recurrent neural networks," 2021, *arXiv:2109.04636*.