Thesis Proposal

Bridging Learning and Logic: Neural Network Controllers Synthesis for Signal Temporal Logic Specifications

Yue Meng

Reliable Autonomous Systems Lab at MIT (REALM)
Department of Aeronautics and Astronautics
Massachusetts Institute of Technology

Feb 14, 2025

Thesis Committee:
Professor Chuchu Fan (Advisor)
Professor Sertac Karaman
Professor Nicholas Roy

External Evaluator:
Professor Glen Chou
Professor Karen Leung

Department Representative:
YYY YYY

Abstract

Learning to plan for complex tasks with temporal dependency and logical constraints is critical to many real-world applications, such as navigation, autonomous vehicles, and industrial assembly lines. For example, the vehicles should make a complete stop before entering the intersection with a stop sign, wait a few seconds, and then drive through it if no other cars are there. And a robot navigating through obstacles to reach the destination should always reach and stay at a charging station for a while to get charged when it is low on battery. These cases require precise reasoning and planning to ensure safety, efficiency, and correctness.

Signal temporal logic (STL) has been widely used to systematically and rigorously specify these requirements. However, traditional methods of generating the control sequence to satisfy STL requirements are computationally complex and not scalable to high-dimensional or systems with complex nonlinear dynamics. Recent works leverage reinforcement learning and imitation learning to solve STL problems. However, they either require delicate reward design state-space augmentation, or post-hoc adjustments, limiting themselves to handling more generalized and diverse STL syntax.

In this thesis, I propose to use data-driven techniques to efficiently and effectively solving general STL problems. The proposed framework exploits the expressiveness of the deep neural networks and provides a systematic symbolic way to construct STL supervisions and encodings to guide the model in learning the tasks. Based upon this framework, I design several instantiations catering to varied assumptions and data availability, leveraging cutting-edge techniques such as differentiable simulation, graph neural networks (GNN), diffusion models, and flow-matching.

To demonstrate the superiority of our methods, we consider a wide range of applications in simulations, including autonomous driving, robotic motion planning, and navigation tasks. Preliminary results show that compared to traditional methods, our approach has huge improvements in computational efficiency and requires less dependency on the system dynamics. Compared to other learning-based methods, the proposed method shows the highest STL satisfaction rate. For future works, I plan to leverage Large Language Models (LLM) to enable open-vocabulary logical reasoning and task planning for more general STL problems.

Contents

1 Introduction

Learning to control a robot to satisfy long-term and complex safety requirements and temporal specifications is critical in autonomous systems and artificial intelligence. For example, the vehicles should make a complete stop before entering the intersection with a stop sign, wait a few seconds, and then drive through it if no other cars are there. And a robot navigating through obstacles to reach the destination should always reach and stay at a charging station for a while to get charged when it is low on battery.

However, designing the controller to satisfy those specifications is challenging. Traditional rule-based methods often require expert knowledge and several rounds of trial and error to find the best design to handle the problem. Other learning-based approaches either learn from demonstrations or rewards to find the control policy to satisfy the behavior specification. Those methods need plenty of expert data or great effort in reward design (an improper reward will result in a learned policy to generate unexpected behaviors)

To let the controller satisfy the exact behaviors, another direction of solving this problem is to describe the requirements in temporal logic specifications. In this proposal, we focus on one of the temporal logic specification forms, termed Signal Temporal Logic (STL), since it offers the most expressive framework to handle a wide range of requirements in robotics and cyber-physical systems.

However, it is challenging to solve for STL specifications, due to its non-Markovian nature and lack of efficient decomposition. Synthesis for STL satisfaction is NP-hard [43]. Classical methods for solving STL include sampling-based methods [79, 37], optimization-based methods [70, 43, 73] and gradient-based methods [13] but they cannot balance solution quality and computation efficiency, becoming intractable towards high-dimensional systems or complex STLs.

There is a growing trend to use learning-based methods to satisfy STL specifications [46, 65, 48, 24, 45] using reinforcement learning [56, 3] or imitation learning [63, 32]. Reinforcement learning-based methods [46] can learn the policy to satisfy the STL specifications via hand-crafted or STL-inspired rewards, but might encounter unexpected behaviors due to ambiguity and sparsity in the reward, and often needs state-augmentation to tackle the non-markovian nature of STL. Imitation learning-based methods [86, 19] regularize the pre-trained models with temporal logic guidance at test time, making the generated trajectories heavily constrained by the original data distribution. To the best of our knowledge, there is no existing learning-based model that can easily take a general form of STL as input (condition) to produce satisfiable solution distributions.

In this thesis, we focus on designing a learning-based framework for efficiently and effectively solving general STL problems. Our proposed framework has the capability to leverage the expressiveness of the deep neural networks and provides a systematic symbolic way to construct STL supervisions and encodings to guide the model in learning STL-specified tasks. Based on this framework, we present several algorithm instantiations catering to varied assumptions and data availability. We demonstrate how cutting-edge deep learning techniques such as differentiable simulation, graph neural networks (GNN), diffusion models, and flow-matching can be used in handling STL specifications.

Preliminary results show the computation efficacy, efficiency, and design flexibility of our proposed pipeline. Compared to classical STL baselines, our framework requires fewer assumptions on the system dynamics and can work with general nonlinear systems or STL formulas. Computationally, our method generates the STL-compliant solutions

an order of magnitude faster than traditional approaches. Moreover, compared to other learning-based methods, we acheive the highest STL compliance rate. These findings validate the effectiveness of our proposed design.

For future work, we aim to explore the integration of Large Language Models (LLMs) in STL problem solving process. Specifically, we plan to leverage LLMs for open-vocabulary logic reasoning and compositional task planning, enabling better generalization across diverse STL specifications.

Our thesis proposal is structured as follows. Section 2 provides a comprehensive literature review, highlighting the limitations of existing classical approaches and learning-based approaches for solving STL problems. Section 3 presents the basic concept of STL and imitation learning. Section 4 introduces our proposed learning framework. Sections 5- 8 discuss preliminary results and proposed future works. Section 9 concludes with a summary of contributions and outlines the future research direction to leverage large language models for STL solving.

2 Literature Review

2.1 Temporal Logic Overview

Temporal logic (or "tense logic"), originated from Philosophy [64] and developed in computer science [62], provides a rigorous framework for reasoning about time-dependent behaviors. Temporal logic allows systems to be analyzed over sequences of events, which is crucial for computer programs and control systems.

Existing temporal logic specifications can be mainly categorized into linear temporal logic (LTL) [62], computation tree logic (CTL) [11], metric temporal logic (MTL) [42], and signal temporal logic (STL) [16, 68, 29]. LTL expresses boolean-type constraints composed by logical operators (e.g., "AND", "OR", "NOT") and temporal operators (e.g., "Always", "Eventually", and "Until"). Unlike LTL checking for a single trace, CTL operates on branching-time models, where specifications are reasoned over a tree of possible future executions. Extending from LTL, MTL introduces timed intervals for the temporal operators, enabling precise time-bounded specifications. For example, the MTL formula "Eventually within [2,5] seconds, reach the goal" ensures task completion within a defined time window. STL further generalizes MTL by accommodating continuous-valued signals. Temporal logic (LTL [62], MTL [42], and STL [52]) can express rich and complex robot behaviors and hence is useful for controller verification and synthesis.

2.2 Classical planning methods for Signal Temporal Logic

Plenty of motion planning algorithms have been aiming to fulfill temporal logic specifications. We refer the readers to this survey [61]. Abstraction-based methods [75, 18, 53] emerged the earliest, where an automaton or a graph is constructed and the search-based planning is conducted on this discrete form of abstraction. These methods often require domain knowledge and are challenging to construct automatically, and it will be costly to apply these methods to Signal Temporal Logic.

The optimization-based approach came out for specific dynamics (e.g., linear), where the STL specifications are modeled as linear constraints and the control policy is found via Convex Quadratic Programming [47] or Mixed Integer Programming (MIP) [83]. Though they do not need discretization or domain expertise, the computation cost is still too high and they cannot solve complicated systems. As a remedy, [73] plans feasible line segments and then uses tracking controllers to adapt to complex dynamics, and [85] uses separation principles to boost the MILP-solving process.

To better handle complex dynamics, sampling-based approaches are proposed which are computationally efficient and suitable for real-time applications. Representative works include STyLuS* [36] that uses biased sampling and its concurrent works [79, 78, 38] which use RRT or RRT*. Another line of work to cope with nonlinear dynamics is gradient-based methods. A differentiable measure for MTL satisfaction is developed in [58]. Inspired by it, STL-cg [44] handles backpropagation for (parametrized) STL formulas under machine learning frameworks. The work [59] learns STL for complex satellite mission planning. The work [13] provides a counter-example guided framework to learn robust control policies. However, gradient descent is well-known to be slow and might stuck into the local minimum. Both types of methods in this section may not guarantee the optimality or completeness of the solution.

2.3 Learning-based methods for Signal Temporal Logic

Recently, machine learning has been widely used for temporal logic controller synthesis, where neural networks (NN) or other forms of parametrized policy are trained via model-free reinforcement learning (RL) [1, 46, 4, 30], model-based policy learning [8, 37, 12, 48, 45, 17], and imitation learning [65, 48, 28]. There exists work [77, 33] that can learn one policy to tackle multiple LTL specs. But for STL tasks, less effort has been made to handle general STLs. Most methods focus on tackling a specific STL. If a new STL comes, they must retrain the neural network to learn the corresponding policy. The paper [27] proposes a model-based approach to handle flexible STL forms but requires differentiable environments and only works on simple environments. Other works [86, 19] explore regularizing the pre-trained models with temporal logic guidance at test time, where the generated trajectories are closely constrained by the original demonstration data distribution. Large language models (LLM) have also been found successful in integration with STL motion planning [6], but their application remains limited to linear dynamics and MILP solvers.

3 Preliminaries

3.1 Dynamical systems

Consider a dynamical system

$$\dot{x} = f(x, u) \tag{1}$$

where $x \in \mathbb{R}^n$ denotes the system state and $u \in \mathbb{R}^m$ denotes the control input. The system state can contain both the agent state (x,y coordinates, velocity, heading, etc.) and environment information (obstacles, timer constraints, etc.) if specified. Here, we consider the states and controls at discrete time steps with time horizon T, thus the dynamics can be rewritten as

$$x_{t+1} = x_t + f(x_t, u_t)\Delta t. \tag{2}$$

We assume the system is disturbance-free: given an initial state x_0 and a control sequence $u_0, u_1, ..., u_{T-1}$, following the system dynamics, we can generate a trajectory $x_0, x_1, ..., x_T$. We call this trajectory a trace (or signal) and denote it as s. We are interested in finding control sequences so that the resulting trajectories satisfy certain temporal and logical constraints formally defined in Section 3.2.

3.2 Signal Temporal Logic

An STL formula comprises predicates, logical connectives, and temporal operators [15]. The predicates are of the form $\mu(x) \geq 0$ where $\mu : \mathbb{R}^n \to \mathbb{R}$ is a function with the state x as the input and returns a scalar value. STL formulas are constructed in the Backus-Naur form:

$$\phi ::= \top \mid \mu \geq 0 \mid \neg\phi \mid \phi_1 \wedge \phi_2 \mid \phi_1 \mathrm{U}_{[a,b]} \phi_2 \tag{3}$$

where $\top$ means "true", $\neg$ means "negation", $\wedge$ means "and", U means "until" and $[a, b]$ is the time interval from a to b. Other operators can be written from the elementary operators above, such as "or": $\phi_1 \vee \phi_2 = \neg(\neg\phi_1 \wedge \neg\phi_2)$, "infer": $\phi_1 \implies \phi_2 = \neg\phi_1 \vee \phi_2$, "eventually": $\Diamond_{[a,b]}\phi = \top \mathrm{U}_{[a,b]}\phi$ and "always": $\Box_{[a,b]}\phi = \neg\Diamond_{[a,b]}\neg\phi$. We denote $s, t \models \phi$ if a signal s at time t satisfies an STL formula ϕ. The detailed Boolean semantics in [52] is iteratively defined as:

$$\begin{aligned}
&s, t \models \top && \text{(naturally satisfied)} \\
&s, t \models \mu \geq 0 && \Leftrightarrow \ \mu(s(t)) > 0 \\
&s, t \models \neg\phi && \Leftrightarrow \ s, t \not\models \phi \\
&s, t \models \phi_1 \wedge \phi_2 && \Leftrightarrow \ s, t \models \phi_1 \text{ and } s, t \models \phi_2 \\
&s, t \models \phi_1 \vee \phi_2 && \Leftrightarrow \ s, t \models \phi_1 \text{ or } s, t \models \phi_2 \\
&s, t \models \phi_1 \implies \phi_2 && \Leftrightarrow \ \text{if } s, t \models \phi_1 \text{ then } s, t \models \phi_2 \\
&s, t \models \phi_1 \mathrm{U}_{[a,b]} \phi_2 && \Leftrightarrow \ \exists t' \in [t + a, t + b] \text{ s.t. } s, t' \models \phi_2 \\
& && \quad \text{and } \forall t'' \in [t, t'] \ s, t'' \models \phi_1 \\
&s, t \models \Diamond_{[a,b]}\phi && \Leftrightarrow \ \exists t' \in [t + a, t + b] \ s, t' \models \phi \\
&s, t \models \Box_{[a,b]}\phi && \Leftrightarrow \ \forall t' \in [t + a, t + b] \ s, t' \models \phi
\end{aligned} \tag{4}$$

To measure how well the trace satisfies the STL formula, [16] proposes a quantitative semantics called robustness score ρ: the STL formula is satisfied if $\rho > 0$ and is violated

if $\rho < 0$, and a larger ρ reflects a larger margin of satisfaction. The robustness score is calculated with the following rules:

$$
\begin{aligned}
&\rho(s,t,\top) = 1, \quad \rho(s,t,\mu \geq 0) = \mu(s(t)) \\
&\rho(s,t,\neg\phi) = -\rho(s,t,\phi) \\
&\rho(s,t,\phi_1 \wedge \phi_2) = \min\{\rho(s,t,\phi_1), \rho(s,t,\phi_2)\} \\
&\rho(s,t,\phi_1 \vee \phi_2) = \max\{\rho(s,t,\phi_1), \rho(s,t,\phi_2)\} \\
&\rho(s,t,\phi_1 \implies \phi_2) = \max\{-\rho(s,t,\phi_1), \rho(s,t,\phi_2)\} \\
&\rho(s,t,\phi_1 \mathrm{U}_{[a,b]} \phi_2) = \\
&\qquad \sup_{t' \in [t+a,t+b]} \min\left\{\rho(s,t',\phi_2), \inf_{t'' \in [t,t']} \rho(s,t'',\phi_1)\right\} \\
&\rho(s,t,\Diamond_{[a,b]}\phi) = \sup_{t' \in [t+a,t+b]} \rho(s,t',\phi) \\
&\rho(s,t,\Box_{[a,b]}\phi) = \inf_{t' \in [t+a,t+b]} \rho(s,t',\phi)
\end{aligned}
\tag{5}
$$

3.3 Deep generative models

Diffusion models and flow matching learn a distribution from data samples. They both contain a forward process (for training) and an inverse process (for generating samples). For DDPM diffusion models [31], the data are diffused with Gaussian noise iteratively towards a white noise, and a neural network is trained to predict the noise at different diffusion steps. In the denoising process (inverse process), the samples initialized from the white noise are recovered gradually by removing the "noise" predicted by this neural net. For flow matching [49], in the forward process, the samples are continuously transformed into a Gaussian noise by a vector field, and the neural network learns to predict the vector field. During the inverse process, new samples are generated by integrating the predicted vector field over over time, starting from noise and progressively refining the samples.

4 Problem formulation

Given a system model in Eq. (2), an STL formula ϕ in Eq. (3) and an initial state set $\mathcal{X}_0 \subseteq \mathbb{R}^n$, find a control policy π such that starting from any state $x \in \mathcal{X}_0$, the roll out trajectory $x_0, x_1, ..., x_T$ of the resulting closed-loop system satisfies the STL specification ϕ, i.e., $\forall x \in \mathcal{X}_0$, $s_\pi(x) \models \phi$.

4.1 Proposed STL learning framework

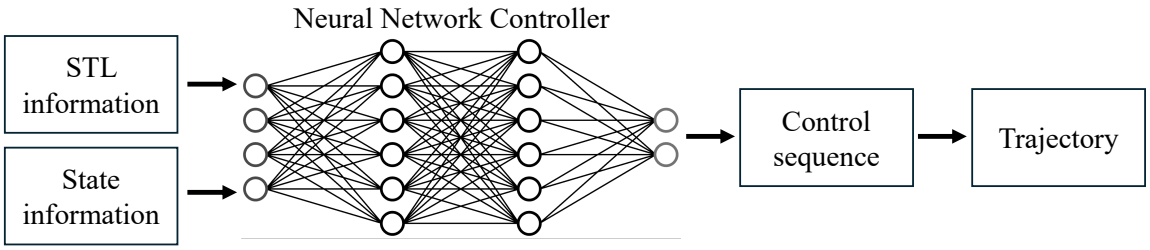


Figure 1: Learning framework for Signal Temporal Logic controller synthesis.

The general learning framework for STL controller synthesis is shown in Fig 1. Here, the inputs to the neural network are the STL information and the state information, and the output to the neural network is a sequence of control actions, which can roll out the trajectory based on the system dynamic. The state information are real-valued variables depicting the robot's state (xy location in the environment, velocity, heading angle, etc.) The STL information can be in varied forms, depending on the specific problem we are trying to solve. For example, it could be real-valued variables representing the environmental variables (obstacles, goal locations). It could also be real-valued variables for different preferences. It could even be a sequence of tokens or a graph representing the STL syntax. We will show the exact form of the STL information in the following sections.

Differentiable	**Demonstration**	**Generalization**	**Method**
✓	✗	N/A	Sec. 5
✓	✓	Low (parameters)	Sec. 6
✗	✓	Medium (structure)	Sec. 7
✗	✗	High (zero-shot)	Sec. 8

Table 1: Different assumptions and expectations for the STL problems.

Based on this proposed framework, we present concrete instantiations for the STL policy learning under varied settings summarized in Table 1. For differentiable simulation environments (the system dynamics and the constraints are differentiable) and a fixed STL, we propose to directly learn a feed-forward neural network controller by maximizing the approximated STL robustness score (Sec. 5; this work [54] was accepted by RA-L and was presented at ICRA2024). With demonstration data available, we learn a diffusion policy that can generate diverse trajectories to satisfy parametrized STLs (Sec. 6; this work [55] was accepted by RA-L and will be presented at ICRA2025). For more

general STL structures and more flexible environments (not necessarily differentiable), we design a graph-encoded flow model that can learn rule-compliant trajectories from demonstrations (Sec. 7; this work is currently under review). Finally, we propose to use large language models (LLM) to learn reward designs for general STLs without any demonstrations (Sec. 8, this is for future works).

5 Differentiable STL Neural Controller Learning

Motivated by the line of work in robustness score [16] and controller synthesis [13], we propose Signal Temporal Logic Neural Predictive Control, which learns a Neural Network controller that can predict a sequence of actions that can generate a trajectory to satisfy the STL. The STL-solving process is conducted in training, and in the deployment phase, we directly use the trained controller to generate STL-satisfied trajectories. Therefore, we don't need any online optimization/searching that is required for those gradient-based or sampling-based methods.

The whole pipeline is as follows: We construct a neural network (NN) controller and sample from the initial state (which might include environment information) distribution. In training, we roll out trajectories using the NN controller. We evaluate the approximated robustness score on those trajectories to maximize the robustness score. In testing, we follow the MPC procedure: our learned controller predicts a trajectory that is safe/rule-satisfied in the short-term horizon, and we pick the first (or first several) actions in the deployment. Finally, when the learned controller is detected violating the STL constraints, a sampling-based backup policy is triggered to at least guarantee the robot's safety.

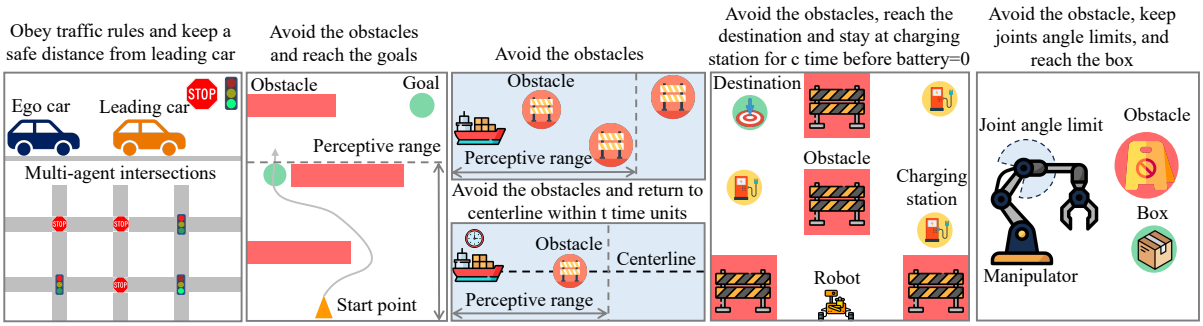


Figure 2: Benchmarks: learning traffic rules, reach-and-avoid game, ship safe / tracking control, navigation and manipulation.

5.1 STL Satisfaction as An Optimization Problem

Based on Sec. 3 and Sec. 4, we further consider a continuous-time hybrid system:

$$\begin{cases} \dot{x} = f(x,u), & x \in \mathcal{C} \\ x^+ = h(x^-), & x \in \mathcal{D} \end{cases} \tag{6}$$

where $x \in \mathbb{R}^n$ denotes the system state and $u \in \mathbb{R}^m$ denotes the control input. The system state here can contain both the agent and environment information. The set $\mathcal{C}$ is the flow set where states follow a continuous flow map and $\mathcal{D}$ is the jump set where states encounter instantaneous changes. A jump captures the scenarios where the agent updates its local observations or resets timers. Like in Sec. 3, we focus on the states and controls at discrete time steps with time horizon T.

To satisfy ϕ, we measure the satisfaction rate of ϕ using the robustness score and thus form the boolean STL satisfaction task as an optimization problem. Denote the policy parametrized with θ as π_θ. For $x \in \mathcal{X}_0$, the policy predicts a sequence of controls: $\pi_\theta(x) = u_{0:T-1}$. Following system dynamics, we can generate the trajectory $s_{\pi_\theta}(x)$ with

horizon $T+1$. Then we compute the robustness score $\rho(s_{\pi_\theta}(x), \phi)$ for the STL formula ϕ on the trajectory $s_{\pi_\theta}(x)$ at time 0 (we omit t for brevity). Our goal is: (omit dynamic constraints)

$$\text{Find } \pi_\theta, \text{s.t. } \rho(s_{\pi_\theta(x)}, \phi) > 0, \forall x \in \mathcal{X}_0 \tag{7}$$

Assuming x follows uniform distribution on $\mathcal{X}_0$, we aim to find the optimal policy which maximizes the expected truncation robustness score:

$$\pi_\theta^* = \arg\max_{\pi_\theta} \mathop{\mathbb{E}}_{x \sim \mathcal{X}_0} \left[\min \left\{ \rho(s_{\pi_\theta(x)}, \phi), \gamma \right\} \right] \tag{8}$$

where $\gamma > 0$ is the truncation factor. The min operator in the expectation encourages the policy to improve "hard" trajectories (with robustness scores $< \gamma$) rather than further increasing "easy" trajectories that already achieve high robustness scores ($\geq \gamma$). This helps achieve high robustness score for all possible cases. In addition, if the optimal value is γ, ϕ is guaranteed to be satisfied on all sampled initial states. In the experiments, we set $\gamma = 0.5$. An ablation study on γ is in Table 2.

5.2 Neural Network Controller Learning for STL Satisfication

We aim to solve Eq. (8) using neural networks. Unlike [13], which solves the STL satisfaction online, we learn the control policy in training, and thus, our approach can run in real-time in testing. We use a fully-connected network (MLP) π_θ to represent the control policy. At each training step, we first sample initial states from $\mathcal{X}_0$ and use π_θ to predict a sequence of actions. Then we roll-out trajectories based on these actions and calculate the robustness score to form a loss function shown in Eq. (8). Finally, we update the parameters of the neural network controller guided by the loss function via stochastic gradient descent. However, there remain two challenges for us to apply the gradient method. First, the hybrid systems in Eq. 6 are non-differentiable at the mode-switching instant. Secondly, ρ is not differentiable due to the max (min) and sup (inf) operators in Eq. (5).

To tackle the non-smoothness in the hybrid systems dynamics, we assume a membership function $I_C : \mathcal{X} \to \mathbb{R}$ exists to imply whether a state x_t is in the flow set ($I_C(x_t) > 0$) or the jump set ($I_C(x_t) < 0$). Thus the forward dynamics can be written as $x_{t+1} = \mathbb{1}\{I_C(x_t) > 0\} f(x_t, u_t)\Delta t + (1 - \mathbb{1}\{I_C(x_t) > 0\})h(x_t)$. Next, we approximate the $\mathbb{1}(I_C(x_t) > 0)$ using $\widetilde{I}_w(x_t) = (1 + \text{Tanh}(w \cdot I_C(x_t)))/2$ with a scaling factor $w > 0$ to control the approximation ($\lim_{w \to \infty} \widetilde{I}_w(x_t) = \mathbb{1}(I_C(x_t) > 0)$). The dynamics now is:

$$x_{t+1} = \widetilde{I}_w(x_t) f(x_t, u_t)\Delta t + (1 - \widetilde{I}_w(x_t))h(x_t) \tag{9}$$

and we can learn STL satisfaction from this hybrid system.

To backpropagate the gradient through STL, we use the approximated robustness score $\tilde{\rho}$ [58], which replaces the max (min) and sup (inf) operators with smooth max (min):

$$\begin{aligned} \widetilde{\max}_k(x_1, x_2, ...) &:= \frac{1}{k} \log \left(e^{kx_1} + e^{kx_2} + \cdots \right) \\ \widetilde{\min}_k(x_1, x_2, ...) &:= -\widetilde{\max}_k(-x_1, -x_2, ...) \end{aligned} \tag{10}$$

where k is a scaling factor for this approximation. If $k \to \infty$, the operator $\widetilde{\max} = \max$ and similarly $\widetilde{\min} = \min$. We use $k = 500$ in our training. An ablation study on k is in Table 2.

Now the framework is differentiable for both the STL robustness score calculation and the system dynamics, we encode the objective in Eq. (8) using the loss function:

$$\mathcal{L}_{\text{STL}} = \frac{1}{|\mathcal{D}_0|} \sum_{x \in \mathcal{D}_0} \max(0, \gamma - \tilde{\rho}(s_{\pi_\theta(x)}, \phi)) \tag{11}$$

where a finite number of states x are uniformly sampled from $\mathcal{X}_0$ to form the training set $\mathcal{D}_0$. Aside from constraint satisfaction, the agent might also need to maximize some performance indices (e.g., "reach the destination as fast as possible.") We hence form the following loss function:

$$\mathcal{L}_{Total} = \mathcal{L}_{Perf} + \lambda \mathcal{L}_{\text{STL}} \tag{12}$$

where $\lambda > 0$ weighs the performance objective $\mathcal{L}_{Perf}$ and the STL violations $\mathcal{L}_{STL}$. We admit the $\lambda \mathcal{L}_{\text{STL}}$ term cannot guarantee the learned policy always satisfy the STL requirement, but empirically we found out this can bring high STL satisfaction rate without much effort in hyperparameter tuning. The $\mathcal{L}_{Perf}$ is often the Euclidean distance between state x_t (in the trajectories starting from $x \sim \mathcal{D}_0$) and goal state x^*, i.e., $\mathcal{L}_{Perf} = \frac{1}{|\mathcal{D}_0|(T+1)} \sum_{x \sim \mathcal{D}_0} |x_t - x^*|$.

5.3 MPC-based Deployment with a Backup Policy

In testing, we follow an online MPC manner. At each time step t, the controller π_θ receives the state x_t and predicts a sequence of commands $u_{0:T-1}$, then we choose the first command u_0 for the agent to execute. Ideally, this policy will satisfy the STL constraints. However, this might not hold in testing due to: (1) imperfect training and (2) out-of-distribution scenario. Thus, we propose a backup policy. We monitor whether the predicted trajectory satisfies the STL specification. If a violation occurs, we sample M trajectories in length $T_0 + 1$ with $T_0 < T$. For the i-th trajectory $\tilde{\xi}_i = \{x_0^i, x_1^i, ...x_{T_0}^i\}$, we evaluate our controller at the final state $x_{T_0}^i$ and rollout a trajectory $\hat{\xi}_i = \{x_{T_0+1}^i, x_{T_0+2}^i ..., x_T^i\}$ (we only keep the first $T - T_0$ timesteps). Since Neural Networks allow batch operations, all the sampled trajectories $\{\tilde{\xi}_i\}_{i=1}^M$ can be efficiently processed in one forward step to get $\{\hat{\xi}_i\}_{i=1}^M$. Finally, we select the trajectory $\xi_i = (\tilde{\xi}_i, \hat{\xi}_i) = \{x_0^i, ..., x_{T_0}^i, x_{T_0+1}^i, ...x_T^i\}$ with the highest robustness score and pick its first action to execute.

If this trajectory still cannot satisfy the STL specification, we choose a trajectory that only satisfies the safety condition:

$$\underset{i}{\text{argmax}}\, \rho(\xi_i, t, \phi), \text{ s.t. } \xi_i, t| = \phi_{\texttt{safe}} \tag{13}$$

where $\phi_{\texttt{safe}}$ contains all the safety constraints in ϕ. We start with $T_0 = 1$, obtain ξ_i using the above method, and gradually increase T_0 until a solution for Eq. (13) is found. If there is still no feasible solution after $T_0 = T$, we choose ξ_i with the longest sub-trajectory starting from x_0^i that satisfies $\phi_{\texttt{safe}}$ and pick the first action to execute. This ensures at least $\phi_{\texttt{safe}}$ are satisfied and the agent can recover to satisfy ϕ in the earliest time. we discretize the action space to L bins, and hence the sampling size is $M = L^{T_0}$. We prove the backup policy has a probabilistic guarantee to find a feasible solution.

Theorem 1 *Assume that a policy u^* exists with a δ-radius neighborhood satisfying constraints ϕ, i.e., $u \models \phi, \forall u \in \{u | \max_{t,i} |u_{t,i} - u_{t,i}^*|_1 \leq \delta\}$, and each step's policy u_t^* is*

uniformly distributed in $\prod_k [u_i^{\texttt{min}}, u_i^{\texttt{max}}]$. *The probability our algorithm finds a solution is:* $\min\{1, (\frac{((2L-4)\delta)^{mT}}{\prod_{i=1}^{m}(u_i^{\texttt{max}}-u_i^{\texttt{min}})^T}\}$.

Proof 1 *We sample from a grid in the policy space at each time step. The probability that a solution can be found is equal to that the* δ*-region contains a grid point at each time step, which is greater than or equal to the probability that the union of the* δ*-hypercube centered at all grids covers the policy* u^*. *The volume of the policy space at each time step is* $\prod_{i=1}^{m}(u_i^{\texttt{max}} - u_i^{\texttt{min}})$. *Each hypercube has a side length* 2δ. *Thus the volume of the union of the hypercubes is greater than (as we omit the cubes that are at the boundary of the policy space)* $(L-2)^m(2\delta)^m$. *Thus the probability of the union covering* u_t^* *is* $\min\{1, \frac{((2L-4)\delta)^m}{\prod_{i=1}^{m}(u_i^{\texttt{max}}-u_i^{\texttt{min}})}\}$, *and for* T *steps, the probability is powered by* T *to derive expected result shown in Theorem 1.*

5.4 Remarks on the STL constraints

To handle different configurations, we augment the system states with additional parameters such as obstacle radius, locations and time constraints. These parameters stay constant unless encountering a reset. The policy learned from this augmentation can solve a family of STL formulas and can adapt to unseen configurations without further fine-tuning.

A wide range of requirements commonly used in robot tasks can be represented by STL, owing to the flexible form of atomic propositions (AP). Denote the 2D location of a robot as $x_{[0:2]} \in \mathbb{R}^2$. We use $\mu(x) = r - ||x_{[0:2]} - x_{\texttt{obs}}||_2$ to check collision with a round obstacle at $x_{\texttt{obs}} \in \mathbb{R}^2$ with radius $r \in \mathbb{R}$, where $||\cdot||_2$ represents the Euclidean norm. For a polygon region $S = \{y : Ay \leq b\}$ with $A \in \mathbb{R}^{k\times 2}$ and $b \in \mathbb{R}^k$, $\mu(x) = b - Ax_{[0:2]}$ can check whether the robot is in S. We use $\mu(x) = -(x_{[0]} - x_{\texttt{min}})(x_{[0]} - x_{\texttt{max}})$ to check whether the agent's x-coordinate is in the interval $[x_{\texttt{min}}, x_{\texttt{max}}]$. Furthermore, we can evaluate whether a system mode has been activated by checking an indicator $I_1 \in \{0, 1\}$ via AP: $\mu(I_1) = I_1 - 0.5$.

However, it is hard to cope with constraints with a time interval. Consider $\phi = \square_{[5,10]}\text{Stay}(A)$ which means *"Always stay at A in the time interval [5,10]"*. One step later, the agent will still try to satisfy $\square_{[5,10]}\text{Stay}(A)$, which actually should be updated to $\square_{[4,9]}\text{Stay}(A)$. Thus we need our policy to be aware of the STL time interval updates. We augment the system state with the timer variables and transform the original STL formula to extra STL constraints on those timer variables. To be more specific, the timer variables follow the dynamic: $\tau_{t+1} = \tau_t + \Delta t$ and the extra STL constraint is $\square_{[0,T]}\left((5 \leq \tau \leq 10) \rightarrow \text{Stay}(A)\right)$.

5.5 Experiments

5.5.1 Experiment setups

Baselines. We compare with RL, model-based RL (MBRL), and classical approaches. For RL, we train Soft Actor Critic [26] [67] under five random seeds with varied rewards. $\mathbf{RL_R}$: uses a hand-crafted reward. $\mathbf{RL_S}$: uses STL robustness score as the reward.

$\mathbf{RL_A}$: uses STL accuracy as the reward. The MBRL baselines are **MBPO**: [35] with STL accuracy as the reward, **PETS**: [9] with a hand-crafted reward, and **CEM**: Cross Entropy Method [14] with STL robustness reward. The rests are **MPC**: Model predictive control via Casadi [2] for nonlinear systems and Gurobi [25] for linear dynamics, **$\mathbf{STL_M}$**: An official implementation for STL-MiLP [73] with a PD control for nonlinear dynamics if needed, and **$\mathbf{STL_G}$**: A gradient-based method similar to [13].

Implementation details. For our method, we set $\gamma = 0.5$, $k = 500$, $T \in [10, 25]$ and $\Delta t \in [0.1\mathbf{s}, 0.2\mathbf{s}]$. Our controller is a three-layer MLP with 256 hidden units in each layer. We uniformly sample 50000 points and train for 50k steps (250k for navigation). We update the controller via the Adam optimizer [40] with a learning rate 3×10^{-4} for most tasks. Training in PyTorch [60] takes 2-12 hours on a V100 GPU.

Metrics. In testing we evaluate the average STL accuracy (the ratio of the short segments starting at each step satisfies the STL) and the computation time. For RL baselines ($\mathbf{RL_R}$, $\mathbf{RL_S}$, $\mathbf{RL_A}$), we also compare the STL accuracy in training.

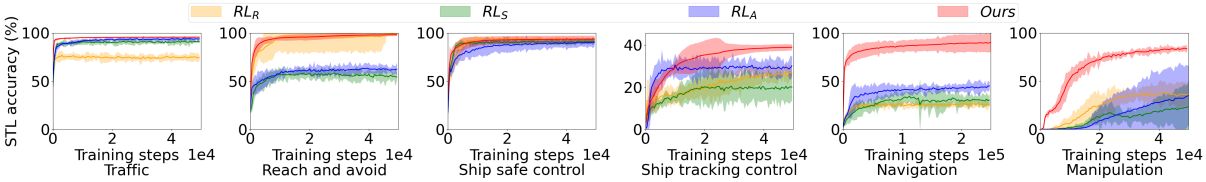


Figure 3: STL accuracy during training

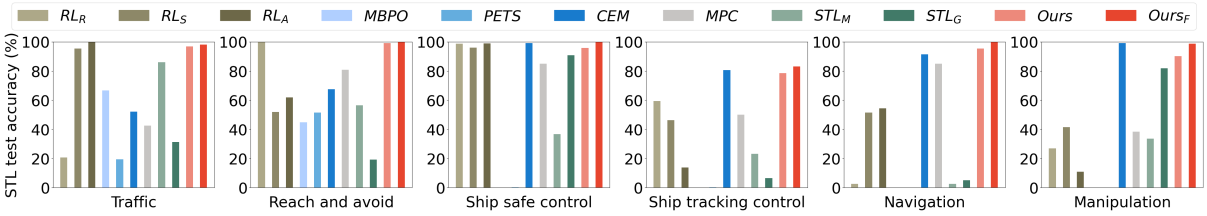


Figure 4: STL accuracy at test phase

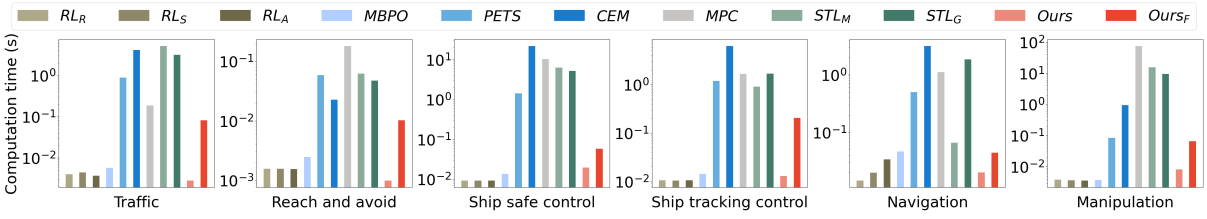


Figure 5: Computation time at test phase

5.5.2 Benchmarks

Driving with traffic rules. We consider driving near intersections where routes and lateral control are provided. The state $(x, v, I_{\texttt{light}}, \tau, \Delta x, v_{\texttt{lead}}, I_{\texttt{yield}})^T$ is for ego car offset, velocity, light indicator (0 for stop sign and 1 for traffic light), timer (stopped time or traffic light phase), leading vehicle distance, its speed, and yield signal (1 for yield and 0 for not). The (partial) dynamics are: $\dot{x} = v$, $\dot{v} = u$, $(\dot{\Delta x}) = v_{\texttt{lead}} - v$, $\dot{\tau} =$

$(1 - I_{\texttt{light}})\mathbb{1}(\text{at stop sign}) + I_{\texttt{light}}$ where $\mathbb{1}(\text{at stop sign}) = \mathbb{1}\left(x(x+1) \leq 0\right)$ as $x = 0$ means being at the intersection, and u is the control. $x, I_{\texttt{light}}, \tau$ will reset at a new intersection, $\Delta x, v_{\texttt{lead}}$ will reset when the leading car changes, and $I_{\texttt{yield}}$ will change when the external yield command is emitted. The rules are: (1) never collide with the leading car, (2) stop by the stop sign for 1 second and then enter the intersection if no yield (3) stop by the intersection if it is red light. Thus, $\Phi = (\neg I_{\texttt{light}} \implies \phi_1) \wedge (I_{\texttt{light}} \implies \phi_2) \wedge \phi_3$ where $(T_{\texttt{total}} = T_r + T_g)$:

$$\begin{aligned}
\phi_1 &= \Diamond_{[0,T]}(\tau > 1) \wedge \left(I_{\texttt{yield}} \implies \Box_{[0,T]}(x < 0)\right) \\
\phi_2 &= \Box_{[0,T]}(\tau \% (T_{\texttt{total}}) > T_r \vee x(x - x_{\texttt{inter}}) > 0) \\
\phi_3 &= \Box_{[0,T]}(\Delta x > 0)
\end{aligned} \tag{14}$$

where T_r and T_g are red and green light phase time, % is modulo operator and $x_{\texttt{inter}}$ is the intersection width. We train π_θ under each sampled scenario (traffic light, stop sign, yield, leading vehicle) at one intersection. In testing, we conduct multi-agent planning with 20 cars and 20 junctions.

Reach-n-avoid game. An agent aims to reach goals and avoid obstacles in a maze shown in Fig. 2(b). It can see up to two levels. The agent state is $(x, v, \Delta y, x_0, l_0, g_0, x_1, l_1, g_1)^T$ where x and v denote the agent's horizontal position and velocity, Δy is the vertical distance to the nearest level above. Here x_0 is the obstacle's leftmost horizontal position, l_0 is the obstacle width, and g_0 is the goal location relative to the obstacle (-1 if no goal). x_1, l_1, g_1 are for the second level above. The agent dynamics are: $\dot{x} = v, \dot{v} = a, (\dot{\Delta y}) = c$ where a is the control and c is a constant. Here $\Delta y, \Delta x_0, l_0, g_0, \Delta x_1, l_1, g_1$ will reset once the agent passes a level. The STL is $\Phi = \phi_1 \wedge \phi_2 \wedge \phi_3 \wedge \phi_4$:

$$\begin{aligned}
\phi_1 &= g_0 > 0 \implies \Diamond_{[0,T]}\left((x - l_0)^2 + \Delta y^2 < r^2\right) \\
\phi_2 &= \Box_{[0,T]}(\Delta y(\Delta y - h) < 0 \implies \Delta x_0(\Delta x_0 - l_0) > 0) \\
\phi_3 &= g_1 > 0 \implies \Diamond_{[0,T]}\left((x - l_1)^2 + (\Delta y - d)^2 < r^2\right) \\
\phi_4 &= \Box_{[0,T]}(\Delta y_1(\Delta y_1 - h) < 0 \implies \Delta x_1(\Delta x_1 - l_1) > 0)
\end{aligned} \tag{15}$$

where r is the goal radius, h is the obstacle's height, $\Delta x_i = x - x_i$, $\Delta y_1 = \Delta y - d$ and d is the gap between two levels. In testing, we control for 500 time steps for evaluation.

Ship collision avoidance. We control a ship (modeled in [21](Sec. 4.2)) to avoid obstacles with varied radii. The 12-dim system state has x, y, ψ, u, v, r to describe the pose and x_1, y_1, r_1 (x_2, y_2, r_2) to denote the (second) closest obstacle with radius r_1 (r_2) and relative position x_1, y_1 (x_2, y_2). The controls are thrust T and rudder angle δ. The dynamics are: $\dot{x} = u\cos\psi - v\sin\psi$, $\dot{y} = u\sin\psi + v\cos\psi$, $\dot{\psi} = r$, $\dot{u} = T$, $\dot{v} = 0.01\delta$, $\dot{r} = 0.5\delta$. The rules are: (1) always be in the river (2) always avoid obstacles. The STL is $\Phi = \phi_1 \wedge \phi_2 \wedge \phi_3$:

$$\begin{aligned}
\phi_1 &= G_{[0,T]}(|y| < D/2) \\
\phi_2 &= G_{[0,T]}((x - x_1)^2 + (y - y_1)^2 \leq r_1^2 \\
\phi_3 &= G_{[0,T]}((x - x_2)^2 + (y - y_2)^2 \leq r_2^2
\end{aligned} \tag{16}$$

where D is the width of the river. In testing, we roll out 20 trajectories for 200 steps to evaluate the performance.

Ship safe centerline tracking. Besides collision avoidance, the ship must also not deviate more than c time units from the centerline. The 10-dim state now has x, y, ψ, u, v, r to denote the ship state, x_1, y_1, r_1 for the closest front obstacle, and τ for the remaining

time the ship can deviate, with $\dot{\tau} = -\mathbb{1}(|y| > \gamma)$ where γ is the deviation threshold. x_1, y_1, r_1 and τ will get reset once the ship passes the current obstacle. The STL is: $\Phi = \phi_1 \wedge \phi_2 \wedge \phi_3$ where,

$$\begin{aligned}
\phi_1 &= G_{[0,T]}(|y| < D/2) \\
\phi_2 &= G_{[0,T]}((x - x_1)^2 + (y - y_1)^2 \leq r_1^2 \\
\phi_3 &= (\tau > 0)\, U_{[0,T]}\left(G_{[0,T]}(|y| < \gamma)\right)
\end{aligned} \tag{17}$$

We rollout 20 trajectories for 200 steps for evaluation.

Robot navigation. A battery-powered robot navigates to reach the destinations and charging stations. The state $(x, y, x_d, y_d, x_c, y_c, \tau_b, \tau_s)^T$ denotes the robot, the target, the charging station, the battery, and the remaining time at the charging station. The controls are speed v and heading θ. The dynamics are: $\dot{x} = v\cos\theta$, $\dot{y} = v\sin\theta, \tau_b = -1$, $\tau_s = -\mathbb{1}\{\text{station}\}$ where its battery will reset: $\tau_b^+ = T$ once it reaches the charger station and the remaining stay time will get reset $\tau_c^+ = c$ once the robot leaves the charging station. The rules are: (1) always avoid obstacles (2) go to the target if high battery (3) if low battery, go to the charging station and stay for c time units and (4) keep the battery level non-negative. The STL is $\Phi = \phi_1 \wedge \phi_2 \wedge \phi_3 \wedge \phi_4 \wedge \phi_5$:

$$\begin{aligned}
&\phi_1 = G_{[0,T]}(\neg\text{In}(\text{Obstacles})) \quad \phi_4 = G_{[0,T]}(\tau_b > 0) \\
&\phi_2 = \tau_b > 1 \implies F_{[0,T]}(\text{Near}(x_d, y_d)) \\
&\phi_3 = \tau_b < 1 \implies F_{[0,T]}(\text{Near}(x_c, y_c)) \\
&\phi_5 = \text{Near}(x_c, y_c) \implies G_{[0,c]}(\text{Near}(x_c, y_c) \vee \tau_s < 0)
\end{aligned} \tag{18}$$

where In(Obstacles) checks if the robot is in any obstacle, and Near(x',y') $= (x - x')^2 + (y - y')^2 \leq r^2$. In testing, we constructed a sequence of destinations and five charging stations. After the robot reaches one destination, the next one will show up. The robot can choose any station for charging.

Manipulation. A 7DoF Franka Emika robot aims to reach the goal without collisions or breaking the joints (We use PyBullet for visualization). The state $(q_0 ... q_6, x, y, z)^T$ denotes the joint angles and the goal location. The dynamics are $\dot{q}_i = u_i, i = 0, ..., 6$ where u_i controls the i-th joint. The obstacle is at $(0.3, 0.3, 0.5)$. The STL is $\Phi = \phi_1 \wedge \phi_2 \wedge ... \phi_9$:

$$\begin{aligned}
\phi_1 &= F_{[0,T]}((x_e - x)^2 + (y_e - y)^2 + (z_e - z)^2 < r_g^2) \\
\phi_2 &= G_{[0,T]}((x_e - 0.3)^2 + (y_e - 0.3)^2 + (z_e - 0.5)^2 > r_o^2) \\
\phi_{i+3} &= G_{[0,T]}(q_i > q_i^{min} \ \wedge \ q_i < q_i^{max}),\ i = 0, ..., 6
\end{aligned} \tag{19}$$

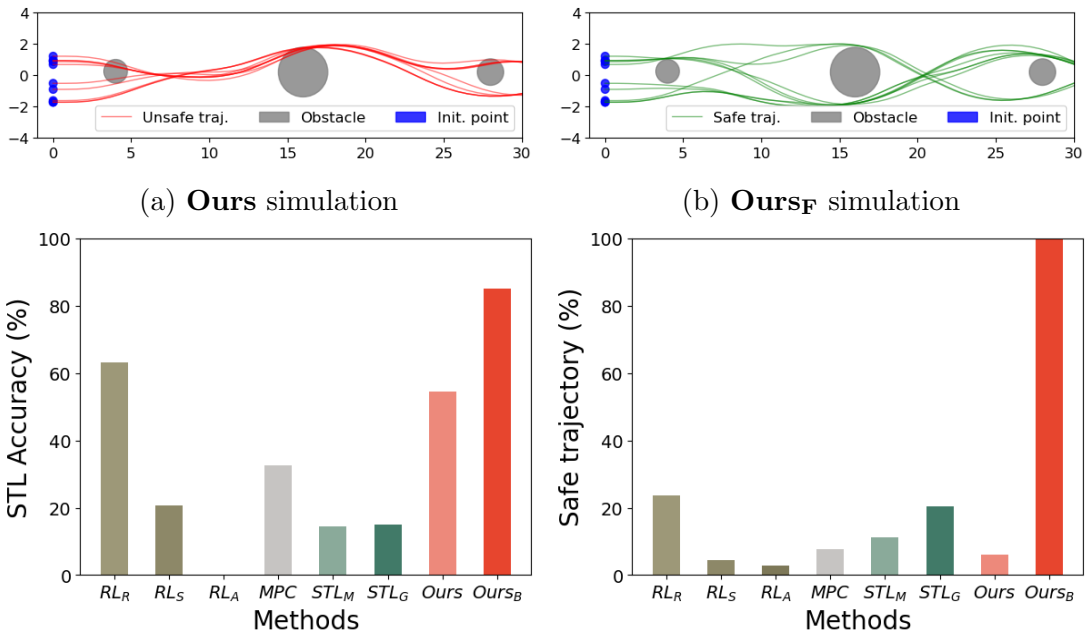
where x_e, y_e, z_e is the end effector, q_i^{min}, q_i^{max} are the joint limits and r_g, r_o are the goal / obstacle radii. In evaluation, the arm is asked to reach a sequence of goals in 250 steps.

5.5.3 Training and testing comparisons

During training, we compare the STL accuracy of our method and the model-free RL baselines ($\mathbf{RL_R}$, $\mathbf{RL_S}$ and $\mathbf{RL_A}$). As shown in Fig. 3, our method in most cases reaches the highest STL accuracy. For tasks with simple dynamics (Traffic, Reach-n-avoid) or simple STL specifications (Ship-safe), the best RL baselines can have similar STL accuracy to ours. However, no one RL baseline can consistently outperform the others. For tasks with moderate system complexity and complicated STL specification (Ship-track,

navigation and manipulation), our approach will have 10% $\sim$ 40% gain in the STL accuracy. We speculate the gain here is because our approach leverages the system dynamics and the gradient information from the STL formula. This shows the advantage of our approach in policy learning compared to RL methods.

In testing, we compare with all baselines and show the result with our backup policy (**Ours$_\mathbf{F}$**). As shown in Fig. 4, aside from **CEM**, **Ours** can outperform the best baselines by 20% for the ship-track task, 45% for the navigation task and 8% for the manipulation task, and only 3% lower than the best approaches on three tasks with simple dynamics or STL formulas. Compared to **CEM**, **Ours** has a slightly lower accuracy on ship safe control, tracking control, and navigation but a much higher accuracy for the rest three tasks. With the backup policy, **Ours$_\mathbf{F}$** achieves the same accuracy as the best RL baselines on Reach-n-avoid and Ship-safe tasks and 1.7% lower on the Traffic task and consistently outperforms **CEM**. The high STL accuracy of **CEM** might be due to the short tasks horizon and low action dimension. The inferior performance for **MPC**, **STL$_\mathbf{M}$ and STL$_\mathbf{G}$** might be because the solvers encounter numerical issues and cannot converge or the optimizer stucks at local minimum. **MBPO** and **PETS** are worse, which might be because they need careful hyper-parameters tuning and learning the dynamics (for the augmented state space) is hard, especially for high-dimension tasks (e.g., navigation and manipulation). As for the computation time, as shown in Fig. 5, **Ours** is on par with RL and 10X-100X faster than classic MPC or STL solvers. While **Ours$_\mathbf{F}$** is slower due to the backup policy, it is still 3X faster than **CEM** and other classical methods.



(a) **Ours** simulation (b) **Ours$_\mathbf{F}$** simulation

(c) STL Accuracy Comparision (d) Safety Rate Comparision

Figure 6: Backup policy in testing

5.5.4 Testing backup policy for out-of-distribution scenarios

When the testing case is out of distribution (OOD), our proposed backup policy can at least maintain the agent's safety and improve the STL accuracy after recovering from the unseen distribution. In the ship-track benchmark, we shift the first obstacle vertically

from the centerline and enlarge the second obstacle, which makes the test case OOD (as in training, the obstacles are smaller and on the centerline). From Fig. 6(c)(d), we can see that without backup policy, **Ours** only achieves 6% safety rate and 54% STL accuracy (from Fig. 6(a) we can see that most of the agents will collide with the first obstacle due to OOD). With the backup policy, $\mathbf{Ours_F}$ achieves 100% safety and increases STL accuracy to 85%. Other baselines are plotted in Fig. 6(c)(d) for reference.

(a) Satisfication threshold γ

γ	Train Acc.	Val. Acc.
0.0	0.855	0.788
0.1	0.920	0.912
0.2	0.918	0.915
0.5	0.957	0.957
0.8	0.957	0.958
1.0	0.958	0.955

(b) Scaling factor k

k	Train Acc.	Val. Acc.
1	0.542	0.542
10	0.950	0.946
100	0.957	0.957
1000	0.958	0.957
10000	0.956	0.956

(c) Neural network size (#neurons x #layers)

Size	Train Acc.	Val. Acc.
32x2	0.951	0.954
32x3	0.951	0.950
64x2	0.951	0.951
64x3	0.951	0.956
128x2	0.954	0.949
128x3	0.956	0.952
256x2	0.952	0.951
256x3	0.957	0.957

(d) Training samples N

N	Train Acc.	Val. Acc.
200	0.800	0.663
400	0.755	0.651
800	0.941	0.836
1600	0.947	0.874
3200	0.952	0.903
6400	0.962	0.920
12800	0.957	0.942
25600	0.954	0.953
50000	0.957	0.957

Table 2: Different setups in training under car benchmark.

5.5.5 Ablation studies

Here on the traffic benchmark we show how different parameters and architectures affect the training and the validation accuracy. As shown in Table 2, for hyperparameters such as γ, k and network size, a wide range of valid values can achieve similar performances ($0.5 < \gamma < 1.0$, $10 < k < 10000$, NN from 64x3 to 256x3). As γ, k, the NN size and the training samples increase, the train/validation STL accuracy almost monotonously increases initially and saturates eventually. Another interesting finding is that, with only 800 training samples, we can already achieve 83.6% STL test accuracy, which is slightly higher than $\mathbf{STL_M}$ (83%, the best classical baseline on the Traffic benchmark). This shows that our approach has a high learning efficiency.

5.6 Conclusions

We propose a neural network controller learning framework to fulfill STL specifications in robot tasks. Unlike RL methods, our approach learns the policy directly via gradient

descent to maximize the approximated robustness score. Experimental results show that our approach achieves the highest STL accuracy compared to other approaches. A backup policy is proposed for STL monitoring process and guarantees the basic safety.

As for limitations, our gradient-based method might be stuck at a local minimum. In testing, our learned policy cannot always satisfy the STL due to the approximation for the robustness score and the generalization error. Although we propose a backup policy to tackle it, a more robust and time-efficient approach is needed.

6 Diverse STL learning with diffusion policy

Our proposed method in the previous section only works with a specific STL. In autonomous driving applications, the specifications for shaping driver behaviors can be diverse and multi-modal. To handle diverse STLs, we proposed a parametric-STL approach to flexibly encode traffic rules, augment the dataset, and learn a controllable diffusion policy to balance quality and diversity[1].

The whole pipeline is: We first specify rules via parameter-STL and use demonstrations to calibrate the parameters. The parameters involve both discrete and continuous values, adding the capacity to form multi-modal and diverse policy distributions. Based on the STL, the parameters, and the original data, we generate the "multiple-outcome" data via trajectory optimization. Next, we use Denoising Diffusion Probabilistic Model (DDPM [31]) to learn from the augmented data. Finally, different from other diffusion-based policies [86, 7], an additional neural network is designed to regulate the trajectories to be rule-compliant and diverse.

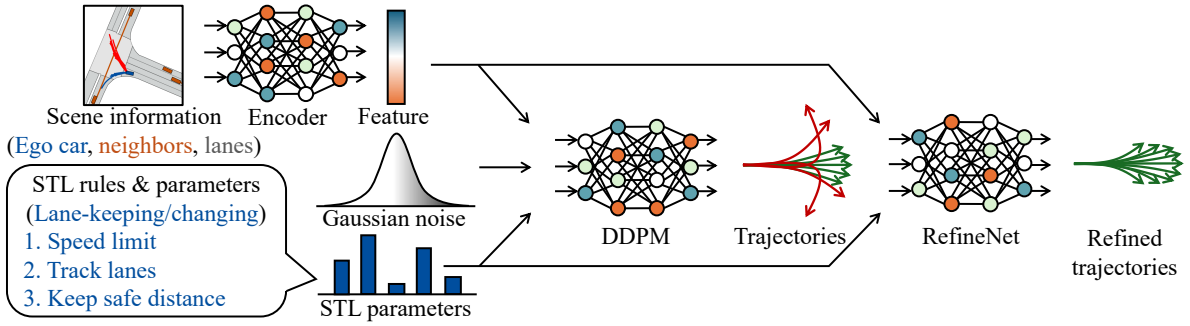


Figure 7: Learning framework. The neural encoder embeds the scene to a feature vector. The DDPM takes the feature vector, the STL parameters (indicating driving modes, speed limit, safe distance threshold, etc) and the Gaussian noise to generate trajectories. RefineNet takes the upstream trajectories and features and generates diverse and rule-compliant trajectories.

6.1 Formulation

Consider an autonomous system, where we denote the state of the agent at time t as $s_t \in \mathcal{S} \subseteq \mathbb{R}^n$, the control $u_t \in \mathcal{U} \subseteq \mathbb{R}^m$, the scene context $c \in \mathcal{C} \subseteq \mathbb{R}^p$, and the STL template $\Psi : \Gamma \times \mathcal{C} \to \Phi$ which generates the STL formula $\phi \in \Phi$ based on the STL parameters $\gamma \in \Gamma$ and the scene provided (i.e., $\phi = \Psi(\gamma, c)$). Assume a known differentiable discrete-time system dynamics: $f : \mathcal{S} \times \mathcal{U} \to \mathcal{S}$, where from s_0 we could generate a trajectory $\tau = (s_0, s_1, ..., s_T) \in \mathbb{R}^{(T+1)\times n}$ based on control sequence $(u_0, u_1, ..., u_{T-1}) \in \mathcal{U}^T$. Assume a known diversity measure $J : \Xi \to \mathbb{R}$ which is a real-valued function over a set of trajectories Ξ.

Given a set of demonstrations $\mathcal{D} = \{(c^i, \tau^i\}_{i=1}^N$ and an STL template Ψ, the goal is first to generate a set of STL parameters $\{\gamma_i\}_{i=1}^N$ such that $\tau_i, 0 \models \Psi(\gamma_i, c_i), \forall i = 1, 2, ...N$, and secondly, learn a policy $\pi : \mathcal{S} \times \mathcal{C} \times \Gamma \to \mathcal{U}^T$ such that for a given initial state s_0, a scene context state c and STL parameters γ, the trajectories set Ξ generated by $\pi(s_0, c, \gamma)$ can both (1) satisfy the STL rule $\tau, 0 \models \Psi(\gamma, c)$, $\forall \tau \in \Xi$, and (2) maximize the diversity

[1] "Diversity" refers to generate **different** trajectories for the **same** STL rule.

measure $J(\Xi)$ (we will use entropy to measure the diversity and the detailed computation for the entropy is shown in Sec. 6.7.2).

6.2 System modeling and STL rules for autonomous driving

Ego car model. We use a unicycle model for the dynamics: $x_{t+1} = x_t + v_t \cos(\theta_t)\Delta t$, $y_{t+1} = y_t + v_t \sin(\theta_t)\Delta t$, $\theta_{t+1} = \theta_t + w_t \Delta t$, $v_{t+1} = v_t + a_t \Delta t$ where the state $s_t = (x_t, y_t, \theta_t, v_t)^T$ stands for ego car's xy coordinates, heading angle, and the velocity at time t, and the controls $u_t = (w_t, a_t)^T$ are angular velocity and acceleration. We denote the ego car width W and length L.

Scene model. The scene context consists of neighbor vehicles and lanes. We consider the N_n nearest neighbors within the R-radius disk centered at the ego vehicle and denote their states at time t as $\mathcal{N} = \{I_t^j, x_t^j, y_t^j, \theta_t^j, v_t^j, L^j, W^j\}_{j=1}^{N_n}$, where the binary indicator I_t^j is one when the j^{th} neighbor is valid and zero otherwise (this happens when there are less than N_n neighbor vehicles within the R-radius disk), and the rests are similar to the ego car state. We represent each lane as $Q = (I, x_1, y_1, \theta_1, ..., x_{N_p}, y_{N_p}, \theta_{N_p}) \in \mathbb{R}^{3N_p+1}$ with an indicator to show its validity and then a sequence of N_p centerline waypoints with 2D coordinates and directions. We denote the current lane, the left adjacent lane, and the right adjacent lane for the ego vehicle as Q_c, Q_l, and Q_r.

STL rules and parameters. We define common STL rules that are required in autonomous driving scenarios,

$$
\begin{aligned}
\phi_0 &= \square_{[0,T]}\, \mathtt{v_{min}} \le \text{Speed}(s) \le \mathtt{v_{max}} \\
\phi_1 &= \square_{[0,T]}\, \text{Dist}(s, \mathcal{N}) \ge \mathtt{d_{safe}} \\
\phi_2 &= \square_{[0,T]}\, \mathtt{d_{min}} \le \text{Dist}(s, Q_c) \le \mathtt{d_{max}} \\
\phi_3 &= \square_{[0,T]}\, |\text{Angle}(s, Q_c)| \le \theta_{\mathtt{max}} \\
\phi_4^H &= \Diamond_{[0,T]} \square_{[0,T]} \mathtt{d_{min}} \le \text{Dist}(s, Q_H) \le \mathtt{d_{max}} \\
\phi_5^H &= \Diamond_{[0,T]} \square_{[0,T]} |\text{Angle}(s, Q_H)| \le \theta_{\mathtt{max}}
\end{aligned}
\tag{20}
$$

where the variables in blue are STL parameters, ϕ_0 is for the speed limit, ϕ_1 specifies the distance to the neighbors should always be greater than the threshold. Formulas ϕ_2 and ϕ_3 restrict the vehicle's distance and heading deviation from the current lane, whereas ϕ_4^H and ϕ_5^H are for the left and right adjacent lanes with $H \in \{l, r\}$ to restrict the car to eventually keep the distance/angle deviation from the target lane within the limit [2]. We consider the driving mode $\mathcal{M}$ belongs to one of the high-level behaviors: lane-keeping ($\mathcal{M} = 0$), left-lane-change ($\mathcal{M} = 1$) and right-lane-change($\mathcal{M} = 2$). Denote the STL parameters $\gamma = (\mathcal{M}, v_{min}, v_{max}, d_{safe}, d_{min}, d_{max}, \theta_{max})^T \in \Gamma \subseteq \mathbb{R}^7$. The STL template thus is defined as:

$$
\begin{aligned}
\Psi(\gamma) = &((\mathcal{M} = 0) \Rightarrow (\phi_0 \wedge \phi_1 \wedge \phi_2 \wedge \phi_3)) \\
&\wedge \big((\mathcal{M} = 1) \Rightarrow (\phi_0 \wedge \phi_1 \wedge \phi_4^l \wedge \phi_5^l)\big) \\
&\wedge ((\mathcal{M} = 2) \Rightarrow (\phi_0 \wedge \phi_1 \wedge \phi_4^r \wedge \phi_5^r))
\end{aligned}
\tag{21}
$$

6.3 STL parameters calibration

Given the STL template Ψ and the expert demonstrations $\mathcal{D}$, we find STL parameters γ_i for each trajectory τ_i and the scene c_i such that $\tau_i, 0 \models \Psi(\gamma_i, c_i)$. We develop an

[2] Since we aim to eventually "always" keep it in the limit, the "Always" operator is put inside the "Eventually" scope to realize this behavior.

annotation tool to manually label the high-level behaviors $\mathcal{M}$ for all the trajectories, then based on $\mathcal{M}$, we calibrate the rest of the STL parameters by making the robustness score on the expert trajectories equal to zero. This is done by conducting min/max extraction on the existing measurements: e.g., to find d_{min}, we first check the high-level policy, then compute the minimum distance from the trajectory τ to the target lane.

6.4 Diverse data augementation

After obtaining the STL parameters for each scene, we augment the original demonstration by generating diverse behaviors using trajectory optimization. For each scene c_i and parameter γ_i, we formulate the following optimization:

$$\begin{aligned} \underset{u_0,u_1,...,u_{T-1}}{\text{minimize}} \quad & \sigma_+(-\rho((s_0,s_1,...,s_T),0,\Psi(\gamma_i,c_i))) \\ \text{subject to} \quad & u_{min} \preccurlyeq u_t \preccurlyeq u_{max}, \forall t=0,1,...,T-1 \\ & s_{t+1}=f(s_t,u_t), \forall t=0,1,...,T-1 \end{aligned} \tag{22}$$

where $\sigma_+(\cdot)=\max(\cdot,0)$, $a \preccurlyeq b$ means the vector a is elementwise no larger than the vector b, and u_{min}, u_{max} are predefined control limits. The system dynamics and STL formula make Eq (22) a nonlinear optimization, and we use a gradient-based method to solve for the solution. To increase solutions' diversity, we consider all three driving modes for $\mathcal{M}$, and for each mode, we run gradient-descent from K initial solutions uniformly sampled from the solution space $\mathcal{U}^T$. We denote our augmented dataset as $\tilde{\mathcal{D}}=\{(\tilde{c}_i,\tilde{\gamma}_i,\{\tilde{\tau}_i^j\}_{j=1}^K)\}_{i=1}^{3N}$ where $\tilde{c}_i=c_{\lfloor i/3 \rfloor}$ and $\tilde{\gamma}_i=\gamma_{\lfloor i/3 \rfloor}$. Here, $\lfloor\cdot\rfloor$ denotes rounding a float number to an integer index.

6.5 Policy learning framework

Given the new dataset $\tilde{\mathcal{D}}$, we learn the diverse and rule-compliant behavior via the learning framework shown in Fig. 7. The encoder network embeds the ego state and the scene to a feature vector. The DDPM network takes the feature vector, STL parameters, and a Gaussian noise to produce trajectories that closely match the distribution in $\tilde{\mathcal{D}}$. Finally, the RefineNet takes upstream features and trajectories to generate diverse and rule-compliant trajectories.

Encoder Network. The ego state and the scene are first transformed to the ego frame, $\tilde{s}$ and $\tilde{c}=\{\tilde{Q}_c,\tilde{Q}_l,\tilde{Q}_r,\tilde{\mathcal{N}}\}$ accordingly. The state $\tilde{s}$ is sent to a fully connected network (FCN) to get ego feature: $z_{ego}=g_{ego}(\tilde{s})\in\mathbb{R}^d$. Similarly, the lanes are fed to a lane FCN to generate feature: $z_{lane}=[g_{lane}(\tilde{Q}_c),g_{lane}(\tilde{Q}_l),g_{lane}(\tilde{Q}_r))]\in\mathbb{R}^{3d}$, where $[\cdot,\cdot...]$ is the vector concatenation. To make the neighbors feature not depend on the neighbor orders, we utilize permutation-invariant operators in [66] with a neighbor FCN g_{nei} to get: $z_{nei}=[\max_j g_{nei}(\tilde{\mathcal{N}}_j),\min_j g_{nei}(\tilde{\mathcal{N}}_j),\sum_j g_{nei}(\tilde{\mathcal{N}}_j)]\in\mathbb{R}^{3d}$, where $\tilde{\mathcal{N}}_j$ is the j$^{\text{th}}$ neighbor feature. The final merged embedding is: $z=[z_{ego},z_{lane},z_{nei}]\in\mathbb{R}^{7d}$.

DDPM Network. Given the embedding z, the STL parameters γ, the sample $\tau^{(0)}$ from $\tilde{\mathcal{D}}$, the random noise ϵ, we generate the diffused samples: $\tau^{(t)}=\sqrt{\bar{\alpha}_t}\tau^{(0)}+\sqrt{1-\bar{\alpha}_t}\epsilon$ for uniformly sampled diffusion steps $t\sim\text{Uniform}(1,T_d)$ and pre-defined coefficients α_t and $\bar{\alpha}_t=\prod_{s=1}^{t}\alpha_s$. The DDPM network $g_d:\mathbb{R}^{7d+7+2T+1}\to\mathbb{R}^{T\times 2}$ takes $z,\gamma,\tau^{(t)},t$ as input and predicts the noise, guided by the diffusion loss in the first stage of the training:

$$\mathcal{L}_d=\mathbb{E}_{\tilde{\mathcal{D}},t}\left[\left|\epsilon-g_d(z,\gamma,\tau^{(t)},t)\right|^2\right] \tag{23}$$

In inference, from the Gaussian noise $\tau^{(T_d)} \sim \mathcal{N}(0, I)$, the trajectories are generated iteratively by the denoising step: $\tau^{(t-1)} = \frac{1}{\sqrt{\alpha_t}}\left(\tau^{(t)} - \frac{1-\alpha_t}{\sqrt{1-\bar{\alpha}_t}} g_d(z, \gamma, \tau^{(t)}, t)\right) + \sigma_t \xi_t$ with $\xi_t \sim \mathcal{N}(0, I)$ and $\sigma_1 = 0$ and $\sigma_t = 1$ for $t \geq 2$. We denote the trajectories generated by DDPM as τ_d.

Refine Network. After DDPM is trained, we use RefineNet, a fully-connected network $g_r : \mathbb{R}^{7d+7+2T} \to \mathbb{R}^{T\times 2}$, to regulate the trajectories generated by the DDPM network to encourage rule-compliance and diversity. RefineNet takes as input the trajectories with the highest STL score from the last five denoising steps and outputs a residual control sequence conditional on the violation of the STL rules, which is:

$$\tau_{final} = \tau_d + \mathbb{1}\{\rho(\tau_d, 0, \Psi(\gamma, c)) < 0\} \cdot g_r(z, \gamma, \tau_d) \tag{24}$$

If the DDPM produced trajectories already satisfy the STL rules, the RefineNet will not affect the final trajectories (i.e., $\tau_{final} = \tau_d$); otherwise, the RefineNet improves trajectories' diversity and the rule satisfaction rate.[3] It is hard to directly optimize for the diversity measure (entropy approximation requires state space discretization, which is non-differentiable). Instead, in the second stage, the RefineNet is updated by the following loss:

$$\mathcal{L}_r = \mathbb{E}_{\tilde{\mathcal{D}}}\left[\mathrm{tr}\left(I - (\mathcal{K}(\{\tau_{final,j}\}_{j=1}^{N_d}) + I)^{-1}\right)\right] \tag{25}$$

where $\mathrm{tr}(\cdot)$ is the matrix trace, and $\mathcal{K} \in \mathbb{R}^{N_d \times N_d}$ is the Direct Point Process (DPP) kernel [84] over N_d samples:

$$\mathcal{K}_{ij} = \mathbb{1}(\rho(\tau_i) \geq 0) \cdot \exp(-|\tau_i - \tau_j|^2) \cdot \mathbb{1}(\rho(\tau_j) \geq 0). \tag{26}$$

Minimizing Eq. (25) increases the trajectory cardinality and quality [84], thus increases the diversity and rule satisfaction.

6.6 Guidance-based online policy refinement

In evaluation, our learned policy might violate the STL rules in the unseen scenarios due to the generalization error. Similar to [86], we use the STL guidance to improve the sampling process. For a new state s_0, scene c, the embedding z and parameter γ in testing, we replace the original denoising step to $\tau^{(t-1)} = \frac{1}{\sqrt{\alpha_t}}\tilde{\tau}^{(t)} + \sigma_t \xi_t$, where $\tilde{\tau}^{(t)}$ is initialized as $\tau^{(t)} - \frac{1-\alpha_t}{\sqrt{1-\bar{\alpha}_t}} g_d(z, \gamma, \tau^{(t)}, t)$ and is updated by minimizing $-\rho(\tilde{\tau}^{(t)}, 0, \Psi(\gamma, c))$ via gradient descent. The work [86] conducts multiple guidance steps at every denoising step. In contrast, we find it sufficient to just conduct the guidance step at the last several denoising steps in the diffusion model to accelerate the computation. Although nonlinear optimization does not guarantee optimality, in practice, this method can satisfy the rules with high probability, as shown below.

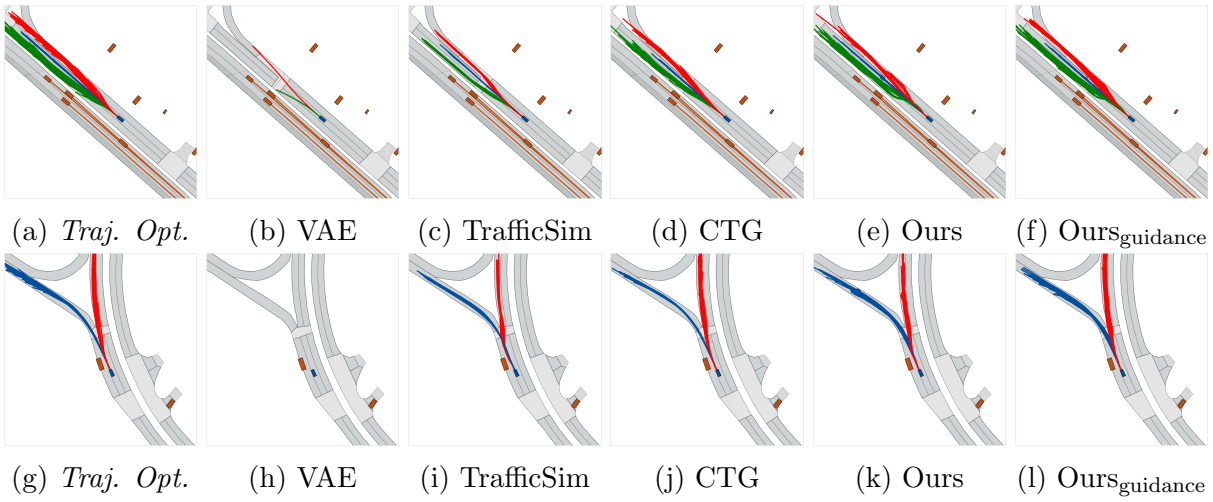
6.7 Experimental results

We first conduct experiments on NuScenes where our method generates the most diverse trajectories quantitatively and visually compared to baselines, reaching the highest STL compliance rate. In closed-loop test we get the lowest collision rate and out-of-lane rate.

[3] Table 3 shows that DDPM results in a very low rule satisfaction rate, which reflects the great potential of using RefineNet for improvement.

Table 3: Open-loop evaluation: The highest is shown in **bold** and the second best is shown in underline. Our data augmentation boosts the diversity for baselines VAE and DDPM. "Ours+guidance" generates the trajectories in the highest quality (Success and Compliance) and diversity (Valid area, Entropy), with the runtime 1/17X to the second best CTG [86].

Methods	Augmentation	Success ↑	Compliance ↑	Valid area ↑	Entropy ↑	Time (s) ↓
Traj. Opt.	Yes	*0.961*	*0.746*	*49.130*	*2.124*	*36.205*
VAE	-	0.337	0.077	0.618	0.162	**0.036**
VAE	Yes	0.253	0.018	0.627	0.200	0.039
DDPM [31]	-	0.514	0.078	1.760	0.455	0.081
DDPM [31]	Yes	0.548	0.050	3.444	0.557	0.081
TrafficSim [74]	Yes	0.699	0.335	6.798	1.059	0.037
CTG [86]	Yes	0.833	0.267	14.933	1.384	13.582
Ours (w/o RefineNet)	Yes	0.624	0.078	5.899	0.780	0.172
Ours ($\mathcal{L}_{STL}$)	Yes	0.817	**0.573**	17.032	1.152	0.174
Ours	Yes	0.782	0.442	20.284	1.411	0.174
Ours+guidance	Yes	**0.840**	0.544	**33.530**	**1.735**	0.786



(a) *Traj. Opt.* (b) VAE (c) TrafficSim (d) CTG (e) Ours (f) $\text{Ours}_{\text{guidance}}$

(g) *Traj. Opt.* (h) VAE (i) TrafficSim (j) CTG (k) Ours (l) $\text{Ours}_{\text{guidance}}$

Figure 8: Open-loop visualizations (Green: "left-lane-change", red: "right-lane-change" and blue: "lane-keeping"). Our approach generates the closest to the *Traj. Opt.* solution and results in the largest trajectory coverage among all the learning methods.

Visualization shows how varied STL parameters affect the agent behavior under the same scene, indicating our approach's potential for diverse agent modeling. We also consider a human-robot scenario, where we generate the most close-to-oracle distribution.

6.7.1 Implementation details

Dataset. NuScenes [5] is a large-scale real-world driving dataset that comprises 5.5 hours of driving data from 1000 scenes (from Boston and Singapore). We use the "trainval" split of the dataset (850 scenes), densely sample from all valid time instants and randomly split the dataset with 70% for training (11763 samples) and 30% for validation (5042 samples). We developed an annotation tool and it took a student four days to label high-level driving behaviors for the data. We mainly focus on vehicles and leave the other road participants (pedestrians and cyclists) for future research.

Algorithm details. The planning horizon is $T = 20$, the duration is $\Delta t = 0.5s$, the number of neighbors is $N_n = 8$, perception radius is $R = 50m$, each lane has $N_p = 15$ waypoints, and the control limits are $u_{max} = -u_{min} = (0.5rad/s, 5.0m/s^2)^T$. In data generation, the number of samples per scene is $K = 64$. Similar to [86], the diffusion steps is $T_d = 100$ and a cosine variance schedule is used. The networks are FCN with 2 hidden layers, with 256 units for each layer and a ReLU activation for the intermediate layers. For our method, we first generate the augmented dataset, then train the DDPM for 500 epochs using Eq. (23). Finally, we freeze the Encoder and DDPM and train the RefineNet for 500 epochs using Eq. (25). We use PyTorch with an ADAM [40] optimizer, a learning rate 3×10^{-4} and a batch size 128. The augmentation takes 5 hours, and training takes 8 hours on an RTX4090Ti GPU.

Table 4: Closed-loop testing: The highest is shown in **bold** and the second best is shown in underline. "Ours+guidance" strikes in diversity and rule compliance with the least collision and out-of-lane rate within an acceptable computation budget.

Methods	Compliance ↑	Valid area ↑	Progress ↑	Collision ↓	Out-of-lane ↓	Time (s) ↓
VAE	0.076	1.403	71.435	0.385	0.154	0.019
DDPM	0.168	5.250	83.493	0.115	**0.000**	0.031
TrafficSim [74]	0.311	3.363	71.113	0.269	0.077	**0.018**
CTG [86]	0.704	16.838	**92.181**	0.115	**0.000**	9.280
Ours	0.448	12.908	74.189	0.192	0.154	0.055
Ours+guidance	**0.763**	**21.577**	88.638	**0.077**	**0.000**	0.379

6.7.2 Open-loop evaluation

Baselines. The methods are ***Traj. Opt.***: Trajectory optimization solution (treated as "Oracle"); **VAE**: Variational Auto-encoder; **DDPM**: Trained with the loss in Eq. (23); **TrafficSim** [74]: train a VAE with an extra rule-violation loss; and **CTG** [86]: train the DDPM and test with guidance during sampling. And **Ours**: our method (Sec. 6.5); **Ours+guidance**: with guidance (Sec. 6.6). For ablations, **Ours (w/o RefineNet)**: no RefineNet; and **Ours ($\mathcal{L}_{STL}$)**: uses $\mathcal{L}_{STL} = \text{ReLU}(0.5 - \rho)$ to train the RefineNet. We implement baselines to accommodate for modality and STL rules, and also train VAE and DDPM on the original NuScenes to show gains from our augmentation.

Metrics. We evaluate the trajectories by (1) **Success**: the ratio of the scenes that have at least one trajectory satisfying the STL rules, (2) **Compliance**: the ratio of generated trajectories satisfying the STL rules (valid trajectories), (3) **Valid area**: The 2d occupancy area of the valid trajectories averaged over all the scenes, (4) **Entropy**: At each time step, we compute the entropy for the normalized angular velocity and the acceleration from the valid trajectories respectively, and average over all time steps and all scenes, and (5) **Time**: measures the trajectory generation time.

Quantitative results. As shown in Table 3, both VAE and DDPM trained on our augmented dataset achieve a higher diversity (valid area and entropy) than them trained on the original NuScenes data, indicating the value of our augmentation technique to generate diverse demonstrations. VAE and DDPM's low compliance rates (less than 10%) imply the need to use an advanced model. Compared to advanced baselines TrafficSim [74] and CTG [86], "Ours" strikes a sharp rise in the quality and diversity: $32-66\%$ higher for rule compliance, $36-198\%$ larger valid area, and up to 33% increase in entropy. Moreover, "Ours+guidance" achieves the highest quality and diversity with 1/17X the time used by the best baseline CTG [4].

Ablation studies. "Ours (w/o RefineNet)" is a bit better than "DDPM", where the gains result from the ensemble of DDPM outputs from the last five denoising steps. With RefineNet and the STL loss used, "Ours ($\mathcal{L}_{STL}$)" gets a $47-188\%$ increase in the diversity measure compared to "Ours (w/o RefineNet)". Further using the diversity loss, "Ours" achieves a $19-22\%$ increase in diversity measure, and "Ours+guidance" generates the most diverse trajectories but at the cost of 4X longer inference time. We can see that adding the RefineNet and using a diversity loss greatly improves the diversity and rule compliance rate (though using the loss $\mathcal{L}_r$ will drop the compliance rate by 3%.) **Visualizations.** In Fig. 8 we plot all the rule-compliant trajectories (generated by different methods) under specific scenes and color them based on high-level driving modes (red for "right-lane-change", blue for "lane-keeping" and green for "left-lane-change"). "Ours" and "Ours+guidance" generate close to *Traj. Opt.* distributions, with the largest area coverage among all learning baselines.

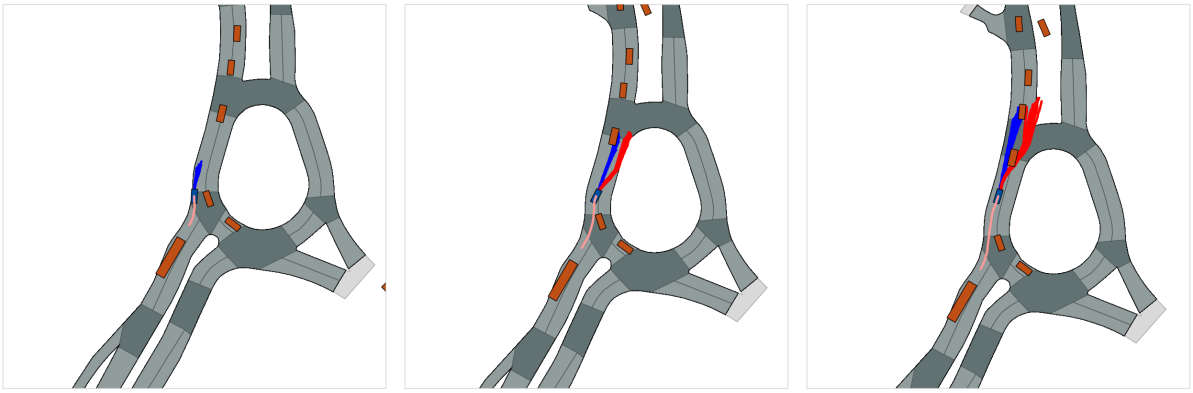
6.7.3 Closed-loop testing

Implementation. We select 26 challenging trials in NuScenes dataset (where the ego car needs to avoid cars on the street or to keep track of curvy lanes), and we set the STL parameters to the minimum/maximum values in the training data to represent the largest feasible range for parameter selection. We start the simulation from these trials and stop it if (1) it reaches the max simulation length or (2) collision happens or the ego car drives out-of-lane. Due to the fast-changing environment, we follow the MPC [54] rather than windowed-policy [7, 71] or other mechanism[5] [69]. At every time step, out of the 64 generated trajectories, we choose the one with the highest robustness score and pick its first action to interact with the simulator. We measure: (1) **Compliance**:

[4] "Ours+guidance" is much faster than CTG because we only use guidance at the last five denoising steps, whereas CTG uses guidance at every step.

[5] The works [7, 71, 69] are mainly for robot manipulation, where the task horizon is long and the environment is relatively static. One challenge in driving scenarios is that the environment can change suddenly in planning (a new neighbor vehicle emerges, lane changes, etc). Thus, the windowed policy might not react to these changes and a mechanism to detect the change and trigger the replanning process is needed. We do not use the goal-conditioned mechanism in [69] as STL cannot be fully conveyed by a few goal states.

the ratio of generated trajectories satisfying the STL rules, (2) **Valid Area**: The 2d occupancy area of the valid trajectories, (3) **Progress**: the average driving distance of the ego car, (4) **Collision**: the ratio of trials ending in collisions, (5) **Out-of-lane**: the ratio of trials ending in driving out-of-lane, and (6) **Time**: the runtime at every step.
Results. As shown in Table 4, our approach without guidance already achieves high performances compared to VAE, DDPM, and TrafficSim [74], with slightly high computation time compared to these learning-based baselines - the overhead in the runtime mainly owes to the DDPM and STL evaluation. Given that the simulation $\Delta t = 0.5s$, this overhead is still in a reasonable range. With the guidance used, "Ours+guidance" surpasses all the baselines in quality and diversity metrics (except for CTG's progress), achieving the lowest collision rate and zero out-of-lane. Compared to the best baseline, CTG, our runtime is just 1/24X of CTG's. This shows our method's ability to provide diverse and high-quality trajectories with an acceptable time budget in tests.



(a) Speed 0 $\sim$ 1m/s (b) Speed 0 $\sim$ 4m/s (c) Speed 0 $\sim$ 6m/s

Figure 9: Diverse behaviors due to varied STL parameters. When the speed limit is low, the agent waits until all vehicles pass the roundabout. When at the middle-speed limit, the agent joins the queue in the middle but yields to other vehicles at high speed. At the high-speed limit, the car joins the queue and keeps its place as traversing the roundabout.

Diverse behaviors under different STL parameters. To show the controllability, we use our method with varied STL parameters to render agent behaviors in a challenging scene shown in Fig. 9, where the ego car waits to join in a roundabout with dense traffic on the right side. We assign three different maximum speed limits to our network, fix the minimum speed to 0m/s, and plot the scenario at t=22. In Fig. 9, the history of the ego car is in pink, and planned trajectories are in blue (for lane-keeping) and red (for right-lane-change). When the max speed limit is low (1m/s), the agent waits until all cars finish the roundabout. When at a speed range [0m/s, 4m/s], the agent joins the queue in the middle but yields to other vehicles at high speed. When at a wider interval ([0m/s, 6m/s]), the car joins the middle of the queue and keeps the place as moving in the roundabout. This finding shows we can model different agent characteristics, which is valuable for realistic agent modeling in simulators.

6.8 Conclusions

We propose a method to learn diverse and rule-compliant agent behavior via data augmentation and Diffusion Models. We model the rules as Signal Temporal Logic (STL),

calibrate the STL parameters from the dataset, augment the data using trajectory optimization, and learn the diverse behavior via DDPM and RefineNet. In the NuScenes dataset, we produce the most diverse and rule-compliant trajectories, with 1/17X the runtime used by the second-best baseline [86]. In closed-loop test, we achieve the highest safety and diversity, and with varied STL parameters we can generate distinct agent behaviors. The limitations are: data annotation effort; it requires differentiable simulation environments; and it cannot handle flexible STL structures. In the next section, we proposed a graph-encoded method to learn solutions for general structured STL tasks.

7 Generalized STL learning with graph-encoded flow

Impeding the advance of general STL policy learning are three critical challenges: (1) most of the papers either work on simplified STLs that are not diverse enough or useful but heavily engineer-designed STLs [51, 54] that are hard to generalize (2) there is no large-scale dataset available to provide paired demonstrations with diverse STL specifications and (3) unlike visual-conditioned or language-conditioned tasks, there lacks an analysis of the effective encoder design to embed the STL information to the downstream neural network. In this section, we tackle all these points above and propose TeLoGraF (Temporal Logic Graph-encoded Flow), a graph-encoded flow matching model that can handle general STL syntax and produce satisfiable trajectories. We first identify four commonly used STL templates and collect over 200K diverse STL specifications. We obtain paired demonstrations using off-the-shelf solvers under each robot domain. Finally, we argue that GNN is a suitable encoder to embed STL information for the downstream tasks and we systematically compare different encoder architectures for STL encoding over varied tasks.

7.1 Formulation

Given a set of demonstrations $\mathcal{D} = \{(\phi_i, \{\tau_{i,k}\}_{k=1}^K)\}_{i=1}^N$ where ϕ_i is the STL formula defined in Sec. 3, and $\{\tau_{i,k}\}_{k=1}^K$ is a batch of trajectories that satisfy the STL specification, $\forall k, \tau_{i,k}, 0 \models \phi_i$, our goal is to learn a conditional generative model $p_\theta(\tau|x_0, \phi)$, so that given a query STL specification ϕ and an initial state x_0, the trajectories sampled from this learned distribution $\tau \sim p_\theta(\cdot|\phi, x_0)$ can satisfy the STL specification, i.e., $\tau, 0 \models \phi$.

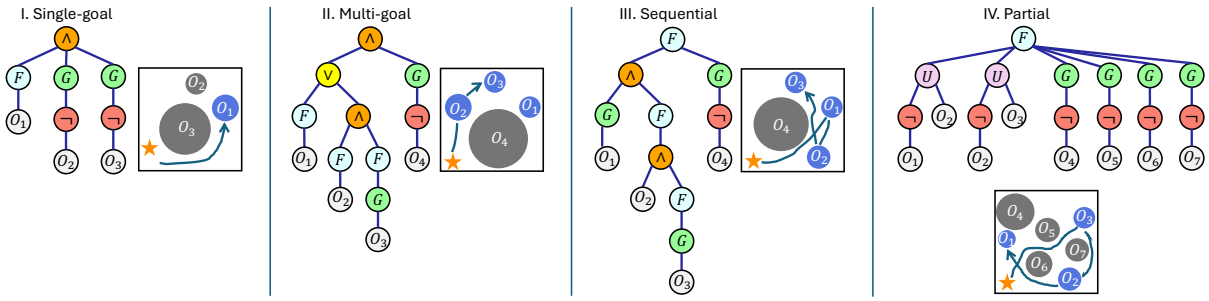


Figure 10: Four STL templates used in this paper. **Single-goal**: reach one goal under time constraints while avoiding obstacles. **Multi-goal**: reach one of the valid subsets of the goals. **Sequential**: All the goals needed to be reached in a strict temporal order. **Partial**: Some goals must be reached first before reaching other goals (no global order is explicitly specified).

7.2 STL specification templates

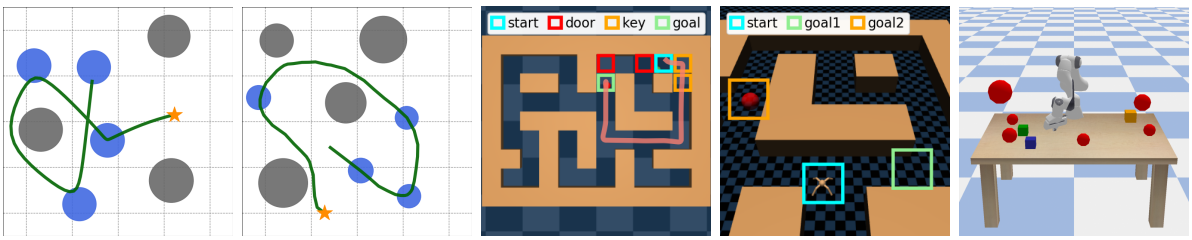
We aim to collect specifications that are both general and representative of the unique characteristics of the STL. Since our focus is primarily on addressing the complexity of STL rather than the skills required to perform each subtask, we restrict the set of atomic propositions semantics to "reach" (an object / a region) and "avoid" (an object / a region), while excluding operations that involve dexterity or agility skills. We want to emphasize the "temporal" and the "logical" facets of the STL specifications, where

the former includes both absolute time constraints or relative temporal ordering (dependencies), and the latter captures patterns to describe multi-choice and selections. In this paper, we consider four types of STL templates: single-goal, multi-goal, sequential, and partial-order, denoted as ϕ_{single}, ϕ_{multi}, $\phi_{sequential}$, and $\phi_{partial}$ in the following Backus-Naur Form (BNF) (similar to Sec. 3, the symbols on the left-hand side of "::=" can take any of the forms split by "|" shown on the right-hand side of the expression):

$$
\begin{aligned}
\phi_{reach} &::= F_{[t_a,t_b]}O \;\mid\; F_{[t_a,t_b]}G_{[t_c,t_d]}O \\
\phi_{avoid} &::= \top \;\mid\; G_{[0,T]}\neg O \;\mid\; \phi_{avoid,1} \wedge \phi_{avoid,2} \\
\phi_{and} &::= \bigwedge_{i=1}^{n_1} \phi_{reach,i} \bigwedge_{j=1}^{n_2} (\phi_{or,j}) \\
\phi_{or} &::= \bigvee_{i=1}^{n_1} \phi_{reach,i} \bigvee_{j=1}^{n_2} (\phi_{and,j}) \\
\phi_{single} &::= \phi_{reach} \wedge \phi_{avoid} \\
\phi_{multi} &::= \phi_{and} \wedge \phi_{avoid} \;\mid\; \phi_{or} \\
\phi_{sequential} &::= \phi_{reach} \;\mid\; F_{[t_a,t_b]}(O \wedge \phi_{sequential}) \wedge \phi_{avoid} \\
\phi_{partial} &::= \bigwedge_{i=1}^{n_1} \neg O_{p_i} U_{[0,T]} O_{q_i} \wedge \bigwedge_{j=1}^{n_2} \phi_{reach,r_j} \wedge \phi_{avoid}
\end{aligned}
\tag{27}
$$

where O_i is the atomic proposition representing "reach the i-th object", and $\neg O$ means "avoid the object". ϕ_{reach} means "eventually reach the object at time interval $[t_a, t_b]$ (and stay there for the time interval $[t_a+t_c, t_b+t_d]$ if $G_{[t_c,t_d]}$ is specified)". ϕ_{avoid} means "always avoid obstacles for all time (here $[0, T]$ denotes the full time horizon)". ϕ_{and} and ϕ_{or} mean "reach a subset of objects" (for example, $\phi_{reach,1} \vee (\phi_{reach,2} \wedge \phi_{reach,3})$ means "reach O_1 or reach both O_2 and O_3").

A graphical illustration for the examples can be found in Figure. 10. The template ϕ_{single} specifies the single-goal-reaching with time constraints. The template ϕ_{multi} specifies a subset of goals to reach. The third template $\phi_{sequential}$ specifies multiple goals that must be reached in a specific order. And the last template $\phi_{partial}$ specifies some objects (O_{p_i}) need to be reached after other objects (O_{q_i}) are reached, and some objects needed to be reached within time constraints (O_{r_j}). All the templates are further paired with ϕ_{avoid} to reflect safety constraints.



(a) Linear (b) Dubins (c) PointMaze (d) AntMaze (e) Franka Panda

Figure 11: Simulation benchmarks. In **Linear** and **Dubins**, a moving robot needs to reach circular regions and avoid circular obstacles. In **PointMaze** and **AntMaze**, the agent needs to reach/avoid square tiles in a maze. In **Franka Panda**, the robot arm needs to reach certain cubes on the table while not colliding with red balls.

7.3 Encoder design

To capture the information from an STL, we treat the STL as a syntax tree, and create a graph node for every operator and atomic proposition node in the syntax tree. The graph nodes are connected based on the edges on the syntax tree, where for every connection, we use a directed edge pointing from the child node to its father node. Mathmatically, given an STL syntax tree, we represent it as a directed graph[6] $G = (V, E, H) \in \mathcal{G}$ with nodes set V, edge set E and node features H (edges pointing from the child node to its father node), where each node feature $h_v \in \mathbb{R}^{d_G}$ contains all the attributes to characterize an STL operator or atomic proposition (AP), including the operator type $I_{type} \in \mathbb{Z}$, start time and end time $t_{start}, t_{end} \in 0, ..., T$ (for operators and APs that do not have the time intervals, we use -1 as the default value), object coordinates x, y, z and radius r (or side length, depending on whether the object is a circle/sphere or a square/cube). Besides, for the Until operator, we need to distinguish its left child with its right child as the order matters, so we use a separate binary variable to indicate whether it is the left child of the Until operator. In total, the node feature dimension is $d_G = 8$.

A multi-layer GNN $\mathcal{F}_\theta : \mathcal{G} \to \mathbb{R}^{d_z}$ operates on this graph data G to get a d_z-dimension embedding. It first iteratively updates the node representations through message passing. The initial node feature for the node v is $h_v^{(0)} = h_v$ discussed above. At each layer l of the GNN, the node v's feature $h_v^{(l)}$ is updated based on the message passing procedure with its neighbors (children) $\mathcal{N}(v)$:

$$h_v^{(l+1)} = \gamma\left(h_v^{(l)}, \bigoplus_{u \in \mathcal{N}(v)} \psi(h_v^{(l)}, h_u^{(l)})\right) \tag{28}$$

here $\gamma(.)$ and ψ are the nonlinear update and message functions to increase the expressiveness of the GNN, and $\oplus$ is the permutation-invariant function (e.g., sum, mean, min, max). After L layers of message passing, the final representation $h_v^{(L)} \in \mathbb{R}^{d_z}$ is read out using another aggregation function $h_G = \bigoplus_{v \in V} h_v^{(L)}$, which will be used for the downstream trajectory generation task.

Expressiveness of GNN to encode STL. It is crucial to verify theoretically that the GNN can distinguish different STLs. The work [82] shows GNN has (at most) the same expressiveness as the 1-dimensional Weisfeiler Leman graph isomorphism test (WL-test). It has been shown in [39] that the WL-test is a complete test for trees, hence GNN can distinguish two STLs (syntax trees) if they are different on the graph level.

7.4 Conditional flow for trajectory generation

Given the STL embedding $z \in \mathbb{R}^{d_z}$ for ϕ and the initial state $x_0 \in \mathbb{R}^n$, we aim to learn a conditional flow neural network $\mathcal{H}_\omega : \mathbb{R}^n \times \mathbb{R}^{d_z} \times \mathbb{R}^1 \times \mathbb{R}^{T \times (m+n)} \to \mathbb{R}^{T \times (m+n)}$ to generate the trajectory $\hat{\tau} \in \mathbb{R}^{T \times (m+n)}$ that can satisfy the STL. In each training step, we randomly draw a demonstration[7] $X_1 = \tau$ with its corresponding STL ϕ from the dataset (x_0 is the first state in τ) and randomly draw X_0 from the Gaussian distribution $\mathcal{N}(0, I)$ and denote $\Delta X = X_1 - X_0$. Then, we uniformly sample $t \sim \text{Uniform}(0, 1)$ and take the linear

[6]Here, we overload the symbol G previously used as the "always" operator for consistency with the normal notation.

[7]We use X_0 to denote the samples from the source distribution, and X_1 for the samples from the target data distribution, while using x for the states in the trajectory.

combination: $X_t = tX_1 + (1-t)X_0$. We denote G as the graph for ϕ as mentioned in Sec. 7.3. To this end, we can formulate the Flow Matching loss:

$$\mathcal{L}_{FM} = \mathbb{E}_{t,X_0,X_1}||\mathcal{H}_\omega\left(\mathcal{F}_\theta(G), x_0, t, X_t\right) - \Delta X||^2 \tag{29}$$

which is used to update the neural network parameters θ and ω in the training. The network $\mathcal{H}_\omega$ learns a velocity field (conditional on ϕ and x_0) which determines a flow $\psi : [0,1] \times \mathbb{R}^{T\times(m+n)} \to \mathbb{R}^{T\times(m+n)}$, depicted as $\frac{d}{dt}\psi(t,X) = \mathcal{H}_\omega(\mathcal{F}_\theta(G), x_0, t, \psi(t,X))$ with initial condition $\psi(0,X) = X$. Therefore, we can generate the target sample by first randomly sampling $X' \sim \mathcal{N}(0,I)$, then solve the ODE with $X = X', t = 0$ until $t = 1$, and the target sample will be $\hat{\tau} = \psi(1,X')$. To solve ODE numerically, we discretize the ODE time domain into N_s steps, and sample t uniformly from $\{0, \frac{1}{N_s}, \frac{2}{N_s}, ..., 1\}$ in training. In testing, from random sample X', we run Euler's method to solve ODE: $\psi(t_{i+1}, X') = \psi(t_i, X') + \frac{\mathcal{H}_\omega(\mathcal{F}_\theta(G), x_0, t_i, \psi(t_i, X'))}{N_s}$, where $t_i = \frac{i}{N_s}$, for $i = 0, 1..., N_s - 1$.

7.5 Experiments

Implementation details. We consider five robot simulation environments, including ZoneEnv in [33] with **Linear** (single-integrator dynamics) and **Dubins** (Car dynamics), **PointMaze**, **AntMaze** [22] and **Franka Panda** robot arm [23]. The trajectory length is 512 for PointMaze and AntMaze and 64 for the other environments. For each robot domain, we start with the four STL templates discussed in Sec. 7.2, and for each template, we randomly generate 10000 STL specifications with 2 to 12 objects (regions) for goals and obstacles, and randomly initialize the agent location in each case. For simulation environments like Linear/Dubins and Franka Panda, we use a gradient-based solver to collect trajectories that maximize the STL robustness scores. For non-differentiable environments like PointMaze and AntMaze, we first plan for waypoints on the grids using A* search algorithm, then we use waypoint tracking controllers to generate the low-level trajectories. For each STL, we generate 2 demonstrations, resulting in 80k trajectories under each robot domain. We use 80% for training and 20% for validation. Our TeLoGraF uses a GNN encoder with GCN layers [41] with 4 hidden layers and 256 units in each layer, and a ReLU activation [57] is used for the intermediate layers. The output embedding dimension is 256. The backbone network follows the UNet architecture used in [34] with two extra input for the STL embedding and the initial states. In the flow matching, we set the flow step $N_s = 100$. The learning pipeline is implemented in Pytorch Geometric [20, 60]. The training is conducted for 1000 epochs with a batch size of 256. We use the commonly used ADAM [40] optimizer with an initial learning rate 5×10^{-4} and a cosine annealing schedule that reduces the learning rate to 5×10^{-5} at the 900-th epochs and then keep it as constant for the rest 100 epochs. We use Nvidia L40S GPUs for the training, where each training job takes 6-24 hours on a single GPU. During evaluation, for each graph, we sample 256 STL specifications from the training and the validation sets for comparison.

Baselines. We compare with both classical and learning-based methods for STL specification planning. For the classical methods[8], we compare with **Grad**: a gradient-based method in [13], **Grad-lite**: using the gradient-based method but with fewer iterations, and **CEM**: a sampling-based method [37]. For the learning-based baselines, we compare with **CTG**: [86], which uses gradient guidance for the diffusion models,

[8]We compare with classical methods only on selected differentiable environments.

and **LTLDoG**: [19], which uses classifier guidance for the diffusion models (we change their LTL classifier to an STL classifier for our tasks). We also consider varied forms of encoder architectures, such as **GRU**: [10], which uses gated recurrent units to encode the sequence of STL formula, similar to [27], **Transformer**: [80], which uses attention-based auto-regressive models to encode the sequence of STL formula similar to GRU, and **TreeLSTM**: [76], which is similar to GNN but instead of synchronized message passing, TreeLSTM updates nodes features layer-by-layer via LSTM in a bottom-up fashion on syntax trees. Our method consists three variations, **TeLoGraD** (Diffusion): which uses GNN as encoder and learns the trajectories via diffusion models, **TeLoGraF**: which uses GNN as encoder and learns the trajectories via flow models, and **TeLoGraF (Fast)**: uses the pretrained TeLoGraF model with less ODE steps to generate samples.

Metrics.. For each STL, we sample 1024 trajectories and pick the one with the highest STL score as the final trajectory. We compute the average STL satisfaction rate for the final trajectories (ratio of final trajectories that satisfy the STL) and the average computation runtime for the trajectory generation. Without special notice, we focused on the validation set STL satisfaction rate because the satisfaction rate in the training split is close to 1.

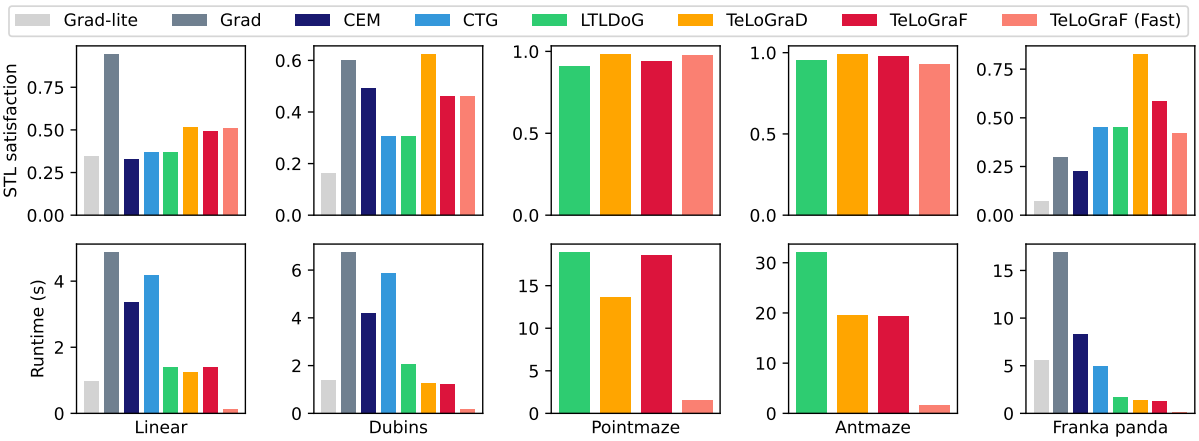


Figure 12: Main results. Our methods outperform both learning-based methods (**CTG** and **LTLDoG**) and classical methods (**Grad**, **CEM**). In four out of the five benchmarks, **TeLoGraD** achieves the highest solution quality. **TeLoGraF (Fast)** achieves the best trade-off between solution quality and efficiency over all benchmarks, especially being 123.6X faster than **Grad** and 60.7X faster than **CEM** with a higher satisfaction rate in the **Franka Panda** environment.

7.5.1 Main results

We first compare our method variations (TeLoGraD, TeLoGraF, and TeLoGraF (Fast)) with classical and learning-based STL planning baselines. As shown in Fig. 12, on most of the environments, our methods achieve a better trade-off between solution quality and runtime. TeLoGraD achieves the highest performance regarding STL satisfaction, with the computation budget lower than almost all the baselines (except Grad-lite on Linear environment, but Grad-lite has low satisfaction rate), which first shows the efficacy of our designed pipeline. The advantage over classical methods increases as we move from low-dimension environments to the high-dimension Franka Panda environment, as

both gradient-based and sampling-based methods struggles to provide high-quality solutions in a high-dimension space. Compared to learning-based methods such as CTG and LTLDoG, our methods do not compute the time-consuming guidance step in the inference stage, hence achieving better efficiency. Interestingly, the results under the maze environments (PointMaze and AntMaze) are very close among these learning-based methods, and the STL satisfaction rate is the highest among all environments. This is probably because the data distribution is limited to the maze layout, making learning easier. Among our methods, TeLoGraF and TeLoGraF(Fast) replace the diffusion models in TeLoGraD with flow-matching models, leading to a performance drop in Dubins and Franka Panda environments, which might be owed to the nonlinear dynamics or forward-kinematics under these environments. However, TeLoGraF (Fast) can achieve on par of TeLoGraF's satisfaction rate with only 1/10 of the runtime, making itself a super-efficient algorithm. On Franka Panda environment, TeLoGraF (Fast) is 123.6X faster than Grad and 60.7X faster than CEM with a higher satisfaction rate than both methods. Overall, the results demonstrate that TeLoGraD provides the best balance between solution quality and computational efficiency, while TeLoGraF (Fast) emerges as a promising alternative for real-time applications for the temporal logic task planning.

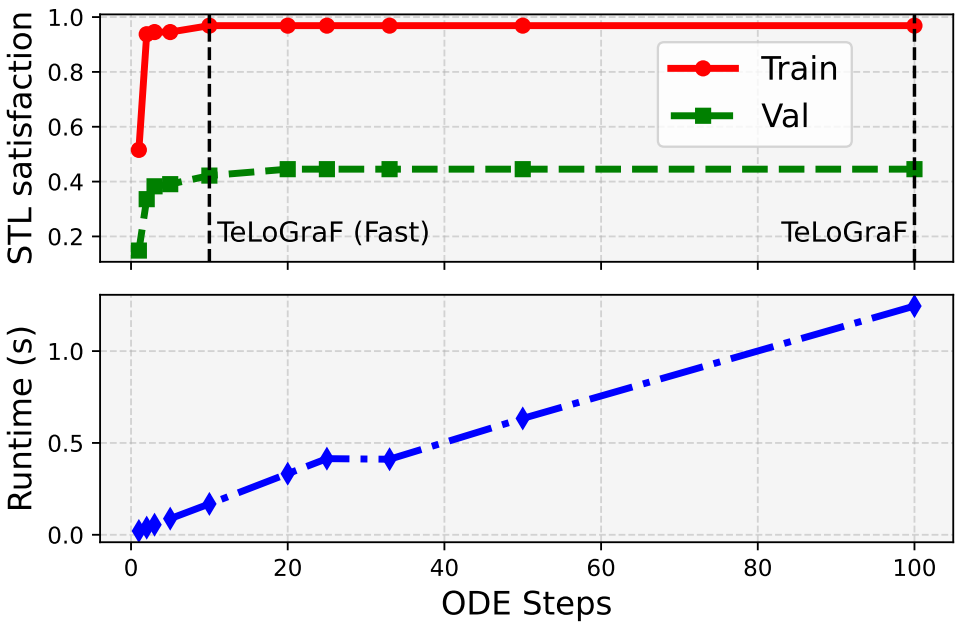


Figure 13: Ablation studies on ODE steps in Dubins environment. TeLoGraF(Fast) achieves a similar STL satisfaction compared to TeLoGraF with only 1/10 of its runtime.

7.5.2 Ablation study on the flow steps

To understand how fast the flow model can be accelerated while maintaining the solution quality, we design a test on the Dubins environment with varied number of flow steps for the sampling. The original TeLoGraF needs to run 100 ODE steps for data generation. Here, we reduce the ODE steps by increasing the ODE time step size[9] in each flow step.

[9]This ODE time step size should be distinguished with the algorithm runtime. The ODE time interval is fixed from 0 to 1 in the flow model sampling process. Thus, the step size in each ODE step determines the number of ODE steps needed.

We evaluate the solution quality by checking its STL satisfaction. As shown in Fig. 13, the algorithm runtime grows linearly with the number of ODE steps. The STL satisfaction rate for the original TeLoGraF is 0.97 on the training split and 0.45 on the validation split. The scores do not drop until the number of ODE steps decreases to 10. Even with 2 ODE steps, the model can obtain an STL satisfaction rate of 0.94 on the training split and 0.34 on the validation. These observations demonstrate the sampling efficiency of the flow models and we hence use 10 ODE steps for "TeLoGraF (fast)".

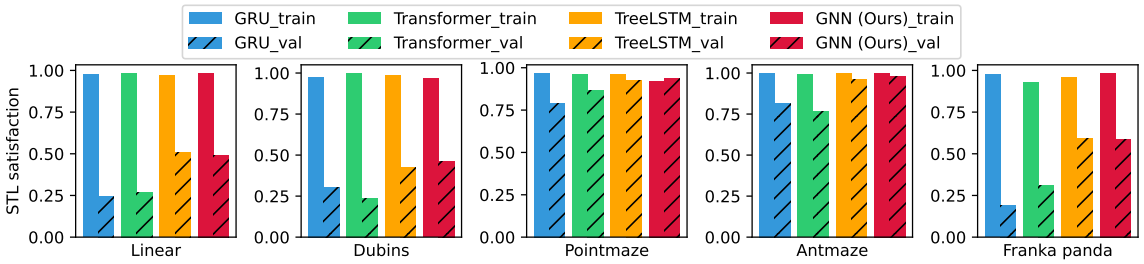


Figure 14: Different encoder architecture comparisons. GNN and TreeLSTM works the best on the validation set.

7.5.3 Ablation study on encoder designs for STL

We compare different architectures to encode STL syntax, including sequence models like GRU and Transformer, and graph-encoding: GNN and TreeLSTM. We use the same pipeline as TeLoGraF, and only change the encoder network. As shown in Fig. 14, all the encoders have a close to 100% satisfaction rate on the training splits, indicating they can distinguish different STL syntax. However, their abilities to generalize to unseen STL syntax are different: GRU and Transformer get lower STL scores than TreeLSTM and GNN; one reason could be that the graph-encoding models inherit permutation-invariance of the syntax tree ("A∨B" is the same as "B∨A"), making the learning more efficient. The performance of TreeLSTM and GNN are similar, and we select GNN encoding owing to its brevity.

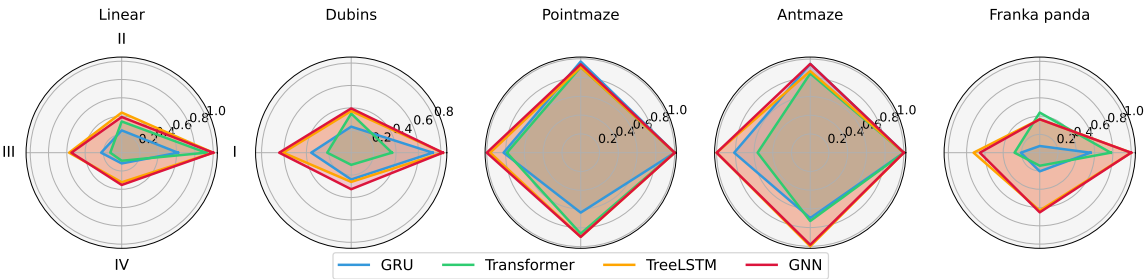


Figure 15: STL satisfaction rate distribution for different categories (I. single-goal; II. multi-goal; III. sequential; IV. partial).

7.5.4 Encoder coverage analysis on varied STL types

We show the validation STL satisfaction over different STL categories in Fig. 15. As expected, most encoders can lead to high satisfaction rate on single-goal STLs. The improvements of the graph-encoding over sequence models are mainly from "Sequential"

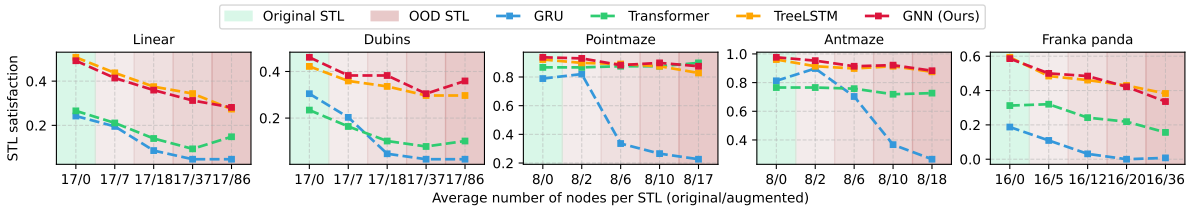


Figure 16: Encoder robustness test under varied STL augmentation (shaded in red). Graph-based encoders (GNN, TreeLSTM) demonstrate greater resilience to synthetic modifications compared to sequence-based models (GRU, Transformer).

and "Partial" types, which implies that, beyond permutation-invariance, graph-encoding methods also excel at propagating time constraints and temporal orderings than the sequence models. However, all models' performance on II-IV STLs is relatively low on Linear, Dubins and Franka Panda, which implies it is challenging for current models to generalize to complex STLs such as multi-goal satisfaction, sequential, or partial constraints. This highlights the value of our dataset in examining the limitation of the current models in handling complex STLs. Future improvements in architectures or training strategies are necessary to foster better temporal logic planning.

7.5.5 Robustness test for encoders on out-domain STLs

We alter the STLs in the validation splits and use the pre-trained models on the original dataset to produce trajectories. We deliberately keep the trajectories fixed and only change the STLs to ensure the backbone is not affected by out-of-distribution trajectories. For each $\wedge$ and $\vee$ operator in the STL syntax tree, we randomly duplicate a number of their children nodes (e.g., "A$\wedge$B" $\rightarrow$ "A$\wedge$B$\wedge$B"). This augmentation will not affect the solutions. Ideally, if the model is robust, the output should have the same STL satisfaction rate. As shown in Fig. 16, all models result in degradation as the number of nodes augmented increases, but the graph-based encoder (GNN, TreeLSTM) is resilient to the augmentation. In most out-domain cases, they can even outperform the sequence models with normal STL inputs. This implies graph-based encoders inherently capture the structural STL more effectively and generalize better under augmentations.

Limitations. First, TeLoGraF is data-driven and does not offer soundness and completeness guarantees as other classical planning methods. Its performance degrades for tasks with complex STL syntax or STL syntax that are heavily out-of-distribution. This limitation is inherent to most learning-based methods, as they rely on the distribution of training data. Trajectory refinement with classical methods could enhance the model's generalization and robustness on these temporal logic tasks. We leave this for future work.

7.6 Conclusion

We propose TeLoGraF, a novel learning-based framework for solving general Signal Temporal Logic (STL) tasks via graph-encoding and flow-matching. TeLoGraF can handle flexible forms of specifications and outperforms existing classical and data-driven baselines while maintaining fast inference speed. Our analysis highlights the value of graph-based STL encoding. Aside from TeLoGraF, we introduce a new dataset for temporal logic planning, addressing the lack of diverse and structured STL benchmarks.

Our proposed method still has several limitations. First, TeLoGraF is data-driven and does not offer soundness and completeness guarantees as other classical planning methods. Its performance degrades for tasks with complex STL syntax or STL syntax that are heavily out-of-distribution. This limitation is inherent to most learning-based methods, as they rely on the distribution of training data. Trajectory refinement with classical methods could enhance the model's generalization and robustness on these temporal logic tasks. We leave this for future work.

8 Future work: Zero-shot STL policy learning via large language models

For future work, I plan to study how to use LLM to learn more general STL tasks. The proposed methods in the previous sections work on scenarios where either the simulation environment and the constraints are differentiable, or the demonstration data is available, and in both cases, the STL specifications and the state space are defined explicitly *a priori*. However, there are cases in which we need to learn policy in non-differentiable environments (such as contact-rich robot physics) or scenarios where the gradient-based method cannot easily find solutions (e.g., long-horizon tasks), and the demonstration data is not available. For general decision-making tasks, Reinforcement Learning [72] is often used to solve for the optimal solutions. However, the temporal logic results in a non-Markovian system; hence, extra effort is needed to augment the original state space and to design proper reward functions. Previous works try to augment the system to be a Markov Decision Process (MDP), using τ-MDP [1] or F-MDP [81] to tackle the problem, but they are either not scalable to long-horizon tasks, or only work under limit types of STLs. Inspired by the recent work [50], we conjecture that LLM has the capability to automatically design the augmented state space and the corresponding reward function for STL tasks. We expect this proposed method can handle more general STL tasks and can ease the design effort.

8.1 Proposed STL-LLM learning framework

The proposed method is illustrated in Fig. 17. The (description of the) original state information and the STL specifications are sent to the LLM in the form of prompt input. Then LLM generates the code for state augmentation to convert the system to an MDP, and correspondingly design the reward function to guide the policy learning process under the reinforcement learning paradigm. After the learning, the policy is expected to roll out the entire trajectory to satisfy the original STL.

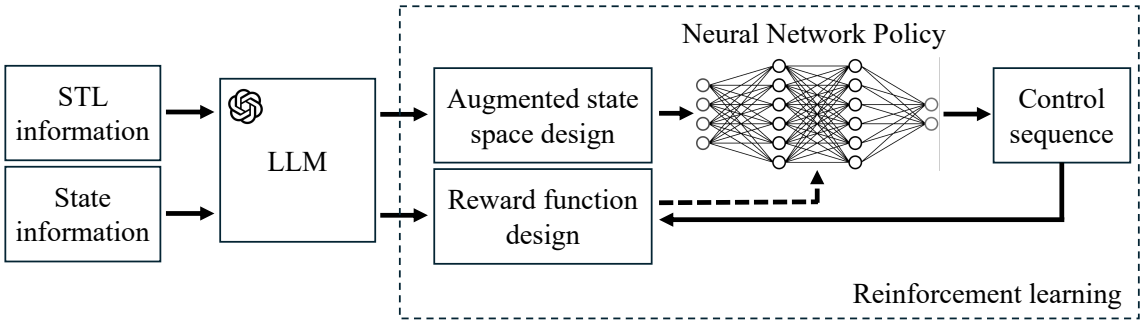


Figure 17: Proposed framework for using LLM to augment the state space and design reward to foster policy learning for general STL specifications.

In this work, we plan to compare with baselines in state-augmentation such as τ-MDP [1] and F-MDP [81], and some other baselines in STL reward designs mentioned in [54], and we expect that our LLM-based design can solve challenging and general STLs more efficiently (using less RL iterations) and effectively (higher success rate for the RL to find STL solutions).

9 Milestones and Schedule

9.1 Classes and Degree Milestones

Here Table 5 summarizes the coursework I have completed, fulfilling all academic requirements for the MIT AeroAstro doctoral program. My major field is in "Autonomous systems", with a minor in "Applied mathematics". Table 6 shows the completed and anticipated degree milestones.

Table 5: Completed coursework satisfying the doctoral program requirements.

Semester	**Courses**	**Type**
Fall 2020	6.867 Machine Learning	Major
Fall 2020	16.413 Principles of Autonomy & Decision Making	Major
Spring 2021	6.832 Underactuated Robotics	Major
Spring 2021	16.s398 Advanced Special Subject in Information and Control	Major
Fall 2021	6.255 Optimization Methods	Minor/Math
Spring 2022	6.7810 Algorithms for Inference	Minor
Spring 2023	6.8200 Sensorimotor Learning	Major
Spring 2023	16.995 Doctoral Research & Communication Seminar	RPC
Fall 2023	16.391 Statistics for Engineers and Scientists	Minor/Math

Table 6: Milestones towards my completion of the doctoral degree.

Completed	
2020/09	Began studies at MIT
2022/01	Completed field evaluation
2023/05	Finished RPC requirement
2024/03	Formed thesis committee
2024/11	Committee meeting #1
Future	
2025/02	Thesis proposal
2025/05	Committee meeting #2
2025/08	Thesis defense

9.2 Research Schedule

My previous and planned thesis research schedule is outlined below.

2022 Fall - 2023 Summer

1. Signal Temporal Logic Neural Predictive Control
   (a) Developed a differentiable controller learning framework for signal temporal logic specifications with backup policy designs.

(b) Revised the manuscripts after 1 round of reviews.

(c) The paper was accepted by RA-L (and was presented at ICRA 2024).

2023 Fall - 2024 Spring

1. Diverse Controllable Diffusion Policy with Signal Temporal Logic

   (a) Developed annotation tools for NuScenes data labeling.

   (b) Augmented the dataset and proposed diverse diffusion policy learning.

   (c) Revised the manuscripts after 2 rounds of reviews.

   (d) The paper was accepted by RA-L (and will be presented at ICRA 2025).

2024 Fall - 2025 Winter

1. TeLoGraF: Temporal Logic Planning via Graph-encoded Flow Matching

   (a) Collected expert demonstrations for varied STL tasks over five different simulation environments.

   (b) Conducted thorough analysis for different encoder architecture designs.

   (c) Submitted the manuscripts to ICML 2025 (under review).

2025 Spring - 2025 Summer (planned)

1. LLM-based State Augmentation and Reward Design for General STL Tasks

   (a) Get familiar with LLM API usage and prompt designs.

   (b) Conduct experiments to showcase the advantage of the proposed method.

   (c) Expected to submit to NeurIPS 2025 or CoRL 2025.

2025 Summer - 2025 Fall (planned)

1. Write Thesis.

2. Defend Thesis and graduate.

References

[1] D. Aksaray, A. Jones, Z. Kong, M. Schwager, and C. Belta. Q-learning for robust satisfaction of signal temporal logic specifications. In *2016 IEEE 55th Conference on Decision and Control (CDC)*, pages 6565–6570. IEEE, 2016.

[2] J. A. Andersson, J. Gillis, G. Horn, J. B. Rawlings, and M. Diehl. Casadi: a software framework for nonlinear optimization and optimal control. *Mathematical Programming Computation*, 11:1–36, 2019.

[3] K. Arulkumaran, M. P. Deisenroth, M. Brundage, and A. A. Bharath. Deep reinforcement learning: A brief survey. *IEEE Signal Processing Magazine*, 34(6):26–38, 2017.

[4] A. Balakrishnan and J. V. Deshmukh. Structured reward shaping using signal temporal logic specifications. In *2019 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 3481–3486. IEEE, 2019.

[5] H. Caesar, V. Bankiti, A. H. Lang, S. Vora, V. E. Liong, Q. Xu, A. Krishnan, Y. Pan, G. Baldan, and O. Beijbom. nuscenes: A multimodal dataset for autonomous driving. In *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, pages 11621–11631, 2020.

[6] Y. Chen, J. Arkin, C. Dawson, Y. Zhang, N. Roy, and C. Fan. Autotamp: Autoregressive task and motion planning with llms as translators and checkers. In *2024 IEEE International conference on robotics and automation (ICRA)*, pages 6695–6702. IEEE, 2024.

[7] C. Chi, Z. Xu, S. Feng, E. Cousineau, Y. Du, B. Burchfiel, R. Tedrake, and S. Song. Diffusion policy: Visuomotor policy learning via action diffusion. *The International Journal of Robotics Research*, page 02783649241273668, 2023.

[8] K. Cho and S. Oh. Learning-based model predictive control under signal temporal logic specifications. In *2018 IEEE International Conference on Robotics and Automation (ICRA)*, pages 7322–7329. IEEE, 2018.

[9] K. Chua, R. Calandra, R. McAllister, and S. Levine. Deep reinforcement learning in a handful of trials using probabilistic dynamics models. *Advances in neural information processing systems*, 31, 2018.

[10] J. Chung, C. Gulcehre, K. Cho, and Y. Bengio. Empirical evaluation of gated recurrent neural networks on sequence modeling. *arXiv preprint arXiv:1412.3555*, 2014.

[11] E. M. Clarke and E. A. Emerson. Design and synthesis of synchronization skeletons using branching time temporal logic. In *Workshop on logic of programs*, pages 52–71. Springer, 1981.

[12] M. H. Cohen and C. Belta. Model-based reinforcement learning for approximate optimal control with temporal logic specifications. In *Proceedings of the 24th International Conference on Hybrid Systems: Computation and Control*, pages 1–11, 2021.

[13] C. Dawson and C. Fan. Robust counterexample-guided optimization for planning from differentiable temporal logic. In *2022 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 7205–7212. IEEE, 2022.

[14] P.-T. De Boer, D. P. Kroese, S. Mannor, and R. Y. Rubinstein. A tutorial on the cross-entropy method. *Annals of operations research*, 134:19–67, 2005.

[15] A. Donzé, T. Ferrere, and O. Maler. Efficient robust monitoring for stl. In *Computer Aided Verification: 25th International Conference, CAV 2013*, pages 264–279. Springer, 2013.

[16] A. Donzé and O. Maler. Robust satisfaction of temporal logic over real-valued signals. *Formal Modeling and Analysis of Timed Systems*, page 92, 2010.

[17] J. Eappen, Z. Xiong, D. Patel, A. Bera, and S. Jagannathan. Scaling safe multi-agent control for signal temporal logic specifications. *arXiv preprint arXiv:2501.05639*, 2025.

[18] G. E. Fainekos, A. Girard, H. Kress-Gazit, and G. J. Pappas. Temporal logic motion planning for dynamic robots. *Automatica*, 45(2):343–352, 2009.

[19] Z. Feng, H. Luan, P. Goyal, and H. Soh. Ltldog: Satisfying temporally-extended symbolic constraints for safe diffusion-based planning. *arXiv preprint arXiv:2405.04235*, 2024.

[20] M. Fey and J. E. Lenssen. Fast graph representation learning with pytorch geometric. *arXiv preprint arXiv:1903.02428*, 2019.

[21] T. I. Fossen. A survey on nonlinear ship control: From theory to practice. *IFAC Proceedings Volumes*, 33(21):1–16, 2000.

[22] J. Fu, A. Kumar, O. Nachum, G. Tucker, and S. Levine. D4rl: Datasets for deep data-driven reinforcement learning. *arXiv preprint arXiv:2004.07219*, 2020.

[23] C. Gaz, M. Cognetti, A. Oliva, P. R. Giordano, and A. De Luca. Dynamic identification of the franka emika panda robot with retrieval of feasible parameters using penalty-based optimization. *IEEE Robotics and Automation Letters*, 4(4):4147–4154, 2019.

[24] Z. Guo, W. Zhou, and W. Li. Temporal logic specification-conditioned decision transformer for offline safe reinforcement learning. In *Forty-first International Conference on Machine Learning*, 2024.

[25] L. Gurobi Optimization. Gurobi optimizer reference manual, 2021.

[26] T. Haarnoja, A. Zhou, P. Abbeel, and S. Levine. Soft actor-critic: Off-policy maximum entropy deep reinforcement learning with a stochastic actor. In *International conference on machine learning*, pages 1861–1870. PMLR, 2018.

[27] W. Hashimoto, K. Hashimoto, M. Kishida, and S. Takai. Neural controller synthesis for signal temporal logic specifications using encoder-decoder structured networks. *arXiv preprint arXiv:2212.05200*, 2022.

[28] W. Hashimoto, K. Hashimoto, and S. Takai. Stl2vec: Signal temporal logic embeddings for control synthesis with recurrent neural networks. *IEEE Robotics and Automation Letters*, 7(2):5246–5253, 2022.

[29] J. He, E. Bartocci, D. Ničković, H. Isakovic, and R. Grosu. Deepstl: from english requirements to signal temporal logic. In *Proceedings of the 44th International Conference on Software Engineering*, pages 610–622, 2022.

[30] Y. He, P. Liu, and Y. Ji. Scalable signal temporal logic guided reinforcement learning via value function space optimization. *arXiv preprint arXiv:2408.01923*, 2024.

[31] J. Ho, A. Jain, and P. Abbeel. Denoising diffusion probabilistic models. *Advances in neural information processing systems*, 2020.

[32] A. Hussein, M. M. Gaber, E. Elyan, and C. Jayne. Imitation learning: A survey of learning methods. *ACM Computing Surveys (CSUR)*, 50(2):1–35, 2017.

[33] M. Jackermeier and A. Abate. Deepltl: Learning to efficiently satisfy complex ltl specifications. *arXiv preprint arXiv:2410.04631*, 2024.

[34] M. Janner, Y. Du, J. B. Tenenbaum, and S. Levine. Planning with diffusion for flexible behavior synthesis. *arXiv preprint arXiv:2205.09991*, 2022.

[35] M. Janner, J. Fu, M. Zhang, and S. Levine. When to trust your model: Model-based policy optimization. *Advances in neural information processing systems*, 32, 2019.

[36] Y. Kantaros and M. M. Zavlanos. Stylus*: A temporal logic optimal control synthesis algorithm for large-scale multi-robot systems. *The International Journal of Robotics Research*, 39(7):812–836, 2020.

[37] P. Kapoor, A. Balakrishnan, and J. V. Deshmukh. Model-based reinforcement learning from signal temporal logic specifications. *arXiv preprint arXiv:2011.04950*, 2020.

[38] J. Karlsson, F. S. Barbosa, and J. Tumova. Sampling-based motion planning with temporal logic missions and spatial preferences. *IFAC-PapersOnLine*, 53(2):15537–15543, 2020.

[39] S. Kiefer. *Power and limits of the Weisfeiler-Leman algorithm*. PhD thesis, Dissertation, RWTH Aachen University, 2020, 2020.

[40] D. Kingma. Adam: a method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

[41] T. N. Kipf and M. Welling. Semi-supervised classification with graph convolutional networks. *arXiv preprint arXiv:1609.02907*, 2016.

[42] R. Koymans. Specifying real-time properties with metric temporal logic. *Real-time systems*, 2(4):255–299, 1990.

[43] V. Kurtz and H. Lin. Mixed-integer programming for signal temporal logic with fewer binary variables. *IEEE Control Systems Letters*, 6:2635–2640, 2022.

[44] K. Leung, N. Aréchiga, and M. Pavone. Backpropagation for parametric stl. In *2019 IEEE Intelligent Vehicles Symposium (IV)*, pages 185–192. IEEE, 2019.

[45] K. Leung and M. Pavone. Semi-supervised trajectory-feedback controller synthesis for signal temporal logic specifications. In *2022 American Control Conference (ACC)*, pages 178–185. IEEE, 2022.

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

[47] L. Lindemann and D. V. Dimarogonas. Control barrier functions for signal temporal logic tasks. *IEEE control systems letters*, 3(1):96–101, 2018.

[48] W. Liu, N. Mehdipour, and C. Belta. Recurrent neural network controllers for signal temporal logic specifications subject to safety constraints. *IEEE Control Systems Letters*, 6:91–96, 2021.

[49] X. Liu, C. Gong, and Q. Liu. Flow straight and fast: Learning to generate and transfer data with rectified flow. *arXiv preprint arXiv:2209.03003*, 2022.

[50] Y. J. Ma, W. Liang, G. Wang, D.-A. Huang, O. Bastani, D. Jayaraman, Y. Zhu, L. Fan, and A. Anandkumar. Eureka: Human-level reward design via coding large language models. *arXiv preprint arXiv:2310.12931*, 2023.

[51] S. Maierhofer, P. Moosbrugger, and M. Althoff. Formalization of intersection traffic rules in temporal logic. In *2022 IEEE Intelligent Vehicles Symposium (IV)*, pages 1135–1144. IEEE, 2022.

[52] O. Maler and D. Nickovic. Monitoring temporal properties of continuous signals. *Formal Techniques, ModellingandAnalysis of Timed and Fault-Tolerant Systems*, page 152, 2004.

[53] J. McMahon and E. Plaku. Sampling-based tree search with discrete abstractions for motion planning with dynamics and temporal logic. In *2014 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 3726–3733. IEEE, 2014.

[54] Y. Meng and C. Fan. Signal temporal logic neural predictive control. *IEEE Robotics and Automation Letters*, 2023.

[55] Y. Meng and C. Fan. Diverse controllable diffusion policy with signal temporal logic. *IEEE Robotics and Automation Letters*, 2024.

[56] V. Mnih. Playing atari with deep reinforcement learning. *arXiv preprint arXiv:1312.5602*, 2013.

[57] V. Nair and G. E. Hinton. Rectified linear units improve restricted boltzmann machines. In *Proceedings of the 27th international conference on machine learning (ICML-10)*, pages 807–814, 2010.

[58] Y. V. Pant, H. Abbas, and R. Mangharam. Smooth operator: Control using the smooth robustness of temporal logic. In *2017 IEEE Conference on Control Technology and Applications*. IEEE, 2017.

[59] A. Pantazides, D. Aksaray, and D. Gebre-Egziabher. Satellite mission planning with signal temporal logic specifications. In *AIAA SCITECH 2022 Forum*, page 1091, 2022.

[60] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, et al. Pytorch: An imperative style, high-performance deep learning library. *Advances in neural information processing systems*, 32, 2019.

[61] E. Plaku and S. Karaman. Motion planning with temporal-logic specifications: Progress and challenges. *AI communications*, 29(1):151–162, 2016.

[62] A. Pnueli. The temporal logic of programs. In *18th annual symposium on foundations of computer science (sfcs 1977)*, pages 46–57. ieee, 1977.

[63] D. A. Pomerleau. Efficient training of artificial neural networks for autonomous navigation. *Neural computation*, 3(1):88–97, 1991.

[64] M. Prior and A. Prior. Erotetic logic. *The Philosophical Review*, 64(1):43–59, 1955.

[65] A. Puranic, J. Deshmukh, and S. Nikolaidis. Learning from demonstrations using signal temporal logic. In *Conference on Robot Learning*, pages 2228–2242. PMLR, 2021.

[66] C. R. Qi, H. Su, K. Mo, and L. J. Guibas. Pointnet: Deep learning on point sets for 3d classification and segmentation. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 652–660, 2017.

[67] A. Raffin, A. Hill, A. Gleave, A. Kanervisto, M. Ernestus, and N. Dormann. Stable-baselines3: Reliable reinforcement learning implementations. *The Journal of Machine Learning Research*, 22(1):12348–12355, 2021.

[68] V. Raman, A. Donzé, D. Sadigh, R. M. Murray, and S. A. Seshia. Reactive synthesis from signal temporal logic specifications. In *Proceedings of the 18th international conference on hybrid systems: Computation and control*, pages 239–248, 2015.

[69] M. Reuss, M. Li, X. Jia, and R. Lioutikov. Goal-conditioned imitation learning using score-based diffusion policies. *arXiv preprint arXiv:2304.02532*, 2023.

[70] S. Sadraddini and C. Belta. Robust temporal logic model predictive control. In *2015 53rd Annual Allerton Conference on Communication, Control, and Computing (Allerton)*, pages 772–779. IEEE, 2015.

[71] P. M. Scheikl, N. Schreiber, C. Haas, N. Freymuth, G. Neumann, R. Lioutikov, and F. Mathis-Ullrich. Movement primitive diffusion: Learning gentle robotic manipulation of deformable objects. *IEEE Robotics and Automation Letters*, 2024.

[72] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov. Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347*, 2017.

[73] D. Sun, J. Chen, S. Mitra, and C. Fan. Multi-agent motion planning from signal temporal logic specifications. *IEEE Robotics and Automation Letters*, 7(2):3451–3458, 2022.

[74] S. Suo, S. Regalado, S. Casas, and R. Urtasun. Trafficsim: Learning to simulate realistic multi-agent behaviors. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 10400–10409, 2021.

[75] P. Tabuada and G. J. Pappas. Model checking ltl over controllable linear systems is decidable. In *HSCC*, volume 2623, pages 498–513. Springer, 2003.

[76] K. S. Tai, R. Socher, and C. D. Manning. Improved semantic representations from tree-structured long short-term memory networks. *arXiv preprint arXiv:1503.00075*, 2015.

[77] P. Vaezipoor, A. C. Li, R. A. T. Icarte, and S. A. Mcilraith. Ltl2action: Generalizing ltl instructions for multi-task rl. In *International Conference on Machine Learning*, pages 10497–10508. PMLR, 2021.

[78] C. I. Vasile, X. Li, and C. Belta. Reactive sampling-based path planning with temporal logic specifications. *The International Journal of Robotics Research*, 39(8):1002–1028, 2020.

[79] C.-I. Vasile, V. Raman, and S. Karaman. Sampling-based synthesis of maximally-satisfying controllers for temporal logic specifications. In *2017 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 3840–3847. IEEE, 2017.

[80] A. Vaswani. Attention is all you need. *Advances in Neural Information Processing Systems*, 2017.

[81] H. Venkataraman, D. Aksaray, and P. Seiler. Tractable reinforcement learning of signal temporal logic objectives. In *Learning for Dynamics and Control*, pages 308–317. PMLR, 2020.

[82] K. Xu, W. Hu, J. Leskovec, and S. Jegelka. How powerful are graph neural networks? *arXiv preprint arXiv:1810.00826*, 2018.

[83] G. Yang, C. Belta, and R. Tron. Continuous-time signal temporal logic planning with control barrier functions. In *2020 American Control Conference (ACC)*, pages 4612–4618. IEEE, 2020.

[84] Y. Yuan and K. Kitani. Diverse trajectory forecasting with determinantal point processes. *arXiv preprint arXiv:1907.04967*, 2019.

[85] Z. Zhang and S. Haesaert. Modularized control synthesis for complex signal temporal logic specifications. *arXiv preprint arXiv:2303.17086*, 2023.

[86] Z. Zhong, D. Rempe, D. Xu, Y. Chen, S. Veer, T. Che, B. Ray, and M. Pavone. Guided conditional diffusion for controllable traffic simulation. In *2023 IEEE International Conference on Robotics and Automation (ICRA)*, pages 3560–3566. IEEE, 2023.