# Training Agents to Satisfy Timed and Untimed Signal Temporal Logic Specifications with Reinforcement Learning

Nathaniel Hamilton[0000−0002−7147−1964], Preston K Robinette[0000−0002−4906−2179], and Taylor T Johnson[0000−0001−8021−9923]

Vanderbilt University, Nashville TN 37212, USA
{nathaniel.p.hamilton,preston.k.robinette,taylor.johnson}@vanderbilt.edu

**Abstract.** Reinforcement Learning (RL) depends critically on how reward functions are designed to capture intended behavior. However, traditional approaches are unable to represent temporal behavior, such as "do task 1 before doing task 2." In the event they can represent temporal behavior, these reward functions are handcrafted by researchers and often require long hours of trial and error to shape the reward function just right to get the desired behavior. In these cases, the desired behavior is already known, the problem is generating a reward function to train the RL agent to satisfy that behavior. To address this issue, we present our approach for automatically converting timed and untimed specifications into a reward function, which has been implemented as the tool STLGym. In this work, we show how STLGym can be used to train RL agents to satisfy specifications better than traditional approaches and to refine learned behavior to better match the specification.

**Keywords:** Deep Reinforcement Learning · Safe Reinforcement Learning · Signal Temporal Logic · Curriculum Learning.

## 1 Introduction

Reinforcement Learning (RL) and Deep Reinforcement Learning (DRL) are fast-growing fields with growing impact, spurred by success in training agents to beat human experts in games like Go [23], Starcraft [25], and Gran Turismo [28]. These results support the claims from [24] that "reward is enough to drive behavior that exhibits abilities studied in natural and artificial intelligence."

However, traditional reward functions are Markovian by nature; mapping states, or states and actions, to scalar reward values without considering previous states or actions [5]. This Markovian nature is in direct conflict with designing reward functions that describe complex, temporally-extended behavior. For example, the task of opening a freezer door, taking something out, and then closing the freezer door cannot be represented by Markovian reward functions, because the success of taking something out of the freezer is dependent on opening the

freezer door first. This problem also extends to the context of safety-critical systems, where the desired behavior might include never entering some region or responding to a situation within a specified amount of time.

Therefore, if we want to use RL and DRL to solve complex, temporally-extended problems, we need a new way of writing and defining reward functions. This is a challenging problem with growing interest as RL research looks into new ways to formulate the reward function to solve these kinds of problems. The most promising approaches look at using temporal logic to write specifications describing the desired behavior, and then generating complex reward functions that help agents learn to satisfy the specifications. Temporal logics are formalism for specifying the desired behavior of systems that evolve over time [16]. Some approaches, like the one presented in this work, take advantage of quantitative semantics [1, 2, 14], while others construct reward machines that change how the reward function is defined depending on which states have been reached [5, 11–13].

Despite the many successes of these approaches, only one is able to incorporate timing constraints ([2]) and many only work with a few RL algorithms that require researchers to write up the problem in a custom format to work with the implementation provided. By ignoring timing constraints, the approaches leave out reactive specifications where systems need to respond within a specified amount of time, like in power systems.

**Our contributions.** In this work, we introduce our approach, and a tool implementation, STLGym, for training RL agents to satisfy complex, temporally-extended problems with and without timing constraints using RL. To the best of our knowledge, and compared to related works discussed in Section 6, our approach is the first that allows users to train agents to satisfy timed and untimed specifications, evaluate how well their agents satisfy those specifications, and retrain agents that do not already satisfy the specifications. We demonstrate the features of our tool and explore some best practices in five interesting example case studies. Our results show STLGym is an effective tool for training RL agents to satisfy a variety of timed and untimed temporal logic specifications.

## 2 Preliminaries

### 2.1 (Deep) Reinforcement Learning

*Reinforcement Learning* (RL) is a form of machine learning in which an agent acts in an environment, learning through experience to increase its performance based on rewarded behavior. *Deep Reinforcement Learning* (DRL) is a newer branch of RL in which a neural network is used to approximate the behavior function, i.e. policy $\pi$. The environment can be comprised of any dynamical system, from video game simulations ([9, 25, 28]) to complex robotics scenarios ([4, 8, 17]). In this work, and to use our tool STLGym, the environment must be constructed using OpenAI's Gym API [4].

Reinforcement learning is based on the *reward hypothesis* that all goals can be described by the maximization of expected *return*, i.e. the cumulative reward.

During training, the agent chooses an action, $u$, based on the input observation, $o$. The action is then executed in the environment, updating the internal state, $s$, according to the plant dynamics. The agent then receives a scalar $r$, and the next observation vector, $o'$. The process of executing an action and receiving a reward and next observation is referred to as a *timestep*. Relevant values, like the input observation, action, and reward are collected as a data tuple, i.e. *sample*, by the RL algorithm to update the current policy, $\pi$, to an improved policy, $\pi^*$. How often these updates are done is dependent on the RL algorithm.

The return is the sum of all rewards collected over the course of an *episode*. An episode is a finite sequence of states, observations, actions, and rewards starting from an initial state and ending when some terminal, i.e. *done*, conditions are met. In this work, we refer to different elements of the episode by their corresponding timestep, $t$. Thus, $r_t$ is the reward value at timestep $t \in [0, T]$, where $T$ is the final timestep in the episode.

### 2.2 Signal Temporal Logic

Signal Temporal Logic (STL) was first introduced in [16] as an extension of previous temporal logics that allows for formalizing control-theoretic properties, properties of path-planning algorithms, and expressing timing constrains and causality relations.

STL specifications are defined recursively according to the *syntax*:

$$\phi := \psi | \neg\phi | \phi \wedge \varphi | \phi \vee \varphi | F_{[a,b]}\phi | G_{[a,b]}\phi | \phi U_{[a,b]}\psi, \tag{1}$$

where $a, b \in \mathbb{R}_{\geq 0}$ are finite non-negative time bounds; $\phi$ and $\varphi$ are STL formulae; and $\psi$ is a predicate in the form $f(w) < d$. In the predicate, $w : \mathbb{R}_{\geq 0} \to \mathbb{R}^n$ is a signal, $f : \mathbb{R}^n \to \mathbb{R}$ is a function, and $d \in \mathbb{R}$ is a constant. The Boolean operators $\neg$, $\wedge$, and $\vee$ are negation, conjunction, and disjunction respectively; and the temporal operators $F$, $G$, and $U$ refer to *Finally* (i.e. eventually), *Globally* (i.e. always), and *Until* respectively. These temporal operators can be timed, having time boundaries where the specification must be met, or untimed without strict time boundaries.

$w_t$ denotes the value of $w$ at time $t$ and $(w, t)$ is the part of the signal that is a sequence of $w_{t'}$ for $t' \in [t, |w|)$, where $|w|$ is the end of the signal. The propositional semantics of STL are recursively defined as follows:

$$\begin{aligned}
(w,t) &\models (f(w) < d) && \Leftrightarrow && f(w_t) < d, \\
(w,t) &\models \neg\phi && \Leftrightarrow && \neg((w,t) \models \phi), \\
(w,t) &\models \phi \wedge \varphi && \Leftrightarrow && (w,t) \models \phi \text{ and } (w,t) \models \varphi, \\
(w,t) &\models \phi \vee \varphi && \Leftrightarrow && (w,t) \models \phi \text{ or } (w,t) \models \varphi, \\
(w,t) &\models F_{[a,b]}\phi && \Leftrightarrow && \exists t' \in [t+a, t+b] \text{ s.t. } (w,t') \models \phi, \\
(w,t) &\models G_{[a,b]}\phi && \Leftrightarrow && (w,t') \models \phi \ \forall t' \in [t+a, t+b], \\
(w,t) &\models \phi U_{[a,b]}\varphi && \Leftrightarrow && \exists t_u \in [t+a, t+b) \text{ s.t. } (w,t_u) \models \varphi \\
& && && \wedge \forall t' \in [t+a, t_u)(w,t') \models \phi.
\end{aligned}$$

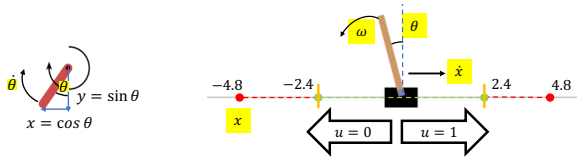
For a signal $(w, 0)$, i.e. the whole signal starting at time 0, satisfying the timed predicate $F_{[a,b]}\phi$ means that "there exists a time within $[a, b]$ such that $\phi$ will eventually be true", and satisfying the timed predicate $G_{[a,b]}\phi$ means that "$\phi$ is true for all times between $[a, b]$". Satisfying the timed predicate $\phi U_{[a,b]}\varphi$ means "there exists a time within $[a, b]$ such that $\varphi$ will be true, and *until* then, $\phi$ is true." Satisfying the untimed predicates have the same description as their timed counterpart, but with $a = 0$ and $b = |w|$.

**Quantitative Semantics** STL has a metric known as *robustness degree* or "degree of satisfaction" that quantifies how well a given signal $w$ satisfies a given formula $\phi$. The robustness degree is calculated recursively according to the *quantitative semantics*:

$$
\begin{aligned}
\rho(w, (f(w) < d), t) &= d - f(w_t), \\
\rho(w, \neg\phi, t) &= -\rho(w, \phi, t), \\
\rho(w, (\phi \wedge \varphi), t) &= \min\big(\rho(w, \phi, t), \rho(w, \varphi, t)\big), \\
\rho(w, (\phi \vee \varphi), t) &= \max\big(\rho(w, \phi, t), \rho(w, \varphi, t)\big), \\
\rho(w, F_{[a,b]}\phi, t) &= \max_{t' \in [t+a, t+b]} \rho(w, \phi, t'), \\
\rho(w, G_{[a,b]}\phi, t) &= \min_{t' \in [t+a, t+b]} \rho(w, \phi, t'), \\
\rho(w, \phi U_{[a,b]}\varphi, t) &= \max_{t_u \in [t+a, t+b]} \Big( \min\{\rho(w, \varphi, t_u), \min_{t' \in [t, t_u)} \big(\rho(w, \phi, t')\big)\}\Big).
\end{aligned}
$$

## 3 Examples

In the remaining sections, we will be referring to these two example RL environments, *Pendulum* and *CartPole*, in order to explain how STLGym works and differs from other approaches. Fig. 1 shows annotated screenshots of the simulated environments.




(a) *Pendulum-v0* (b) *CartPole-v0*

**Fig. 1.** Annotated screenshots showing the simulated environments, *Pendulum* (left) and *CartPole* (right), from the OpenAI Gym benchmarks [4].

### 3.1 Pendulum

The Pendulum environment, shown in Fig. 1.a, consists of an inverted pendulum attached to a fixed point on one side. The agent's goal in this environment is to swing the free end of the pendulum to an upright position, $\theta = 0$, and maintain the position.

The interior plant model changes the state, $s = [\theta, \omega]$, according to the discrete dynamics given the control from the RL agent, $u_t$, in the range $[-2, 2]$ applied as a torque about the fixed end of the pendulum. Additionally, within the environment the pendulum's angular velocity, $\omega$, is clipped within the range $[-8, 8]$, and the angle from upright, $\theta$, is aliased within $[-\pi, \pi]$ radians. $\theta$ is measured from upright and increases as the pendulum moves clockwise. The values $\theta$, $\omega$, and $u$ are used to determine the observation, $o = [\cos(\theta), \sin(\theta), \omega]^T$ and the reward,

$$r_t = -\theta_t^2 - 0.1(\omega_t)^2 - 0.001(u_t)^2. \tag{2}$$

For each episode, the pendulum is initialized according to a uniform distribution with $\theta \in [-\pi, \pi]$ and $\omega \in [-1, 1]$. The episode ends when 200 timesteps have occurred. That means $T$ is always 200.

### 3.2 CartPole

In the CartPole environment[1], a pole is attached to a cart moving along a frictionless track. The agent's goal in this environment is to keep the pole upright, $-12° \leq \theta \leq 12°$, and the cart within the bounds $-2.4 \leq x \leq 2.4$ until the time limit, $t = 200$, is reached. The agent accomplishes this goal by applying a leftward or rightward force to move the cart along the track. The agent's actions are discretized for a "bang-bang" control architecture that moves the cart left when $u = 0$ and right when $u = 1$.

The interior plant model changes the state, $s = [x, \dot{x}, \theta, \dot{\theta}]$, until a terminal condition is met. These terminal conditions are: (1) the cart's position leaves the bounds $-2.4 \leq x \leq 2.4$, (2) the pole's angle is outside the bounds $-12° \leq \theta \leq 12°$, and/or (3) the goal time limit is reached, i.e. $t = 200$.

The original, baseline reward function with this environment gives the agent $+1$ for every timestep the first two terminal conditions are not violated. Thus, the return for an episode is the same as the episode's length. To ensure the agent has a chance to complete at least one timestep successfully, each state element is initialized according to a uniform distribution in the range $[-0.05, 0.05]$. In this implementation, the observation is equivalent to the state, $o = s$.

## 4 Our Approach: STLGym

Our approach focuses solely on augmenting the environment side of the RL process to add an STL monitor and replace the existing reward output with the

[1] The environment is based on the classic cart-pole system implemented for [3], where more information on the dynamics can be found.

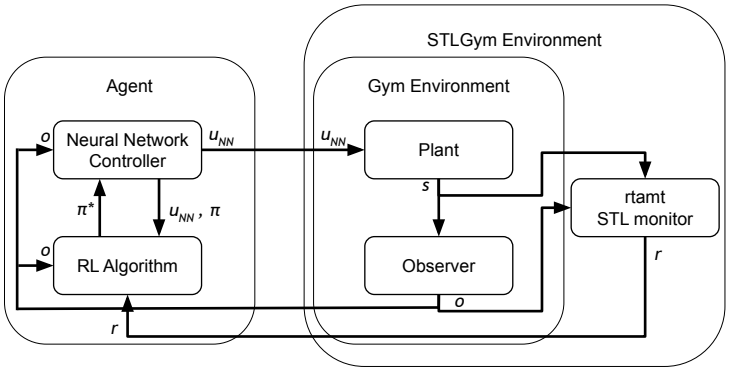



**Fig. 2.** A representation of how STLGym wraps around the user's environment to record signals and replace the reward function.

calculated *robustness degree* as it relates to the desired specification(s), as shown in Fig. 2. This process maintains the standards of the Gym API, so no changes to the RL algorithm are necessary to facilitate its use. As a result, our approach is *algorithm-agnostic*, since no modifications to the RL algorithm are required. Furthermore, since our approach makes use of existing environments, there is great potential for *retraining* learned policies to better optimize satisfying specifications. Our approach is implemented as the tool STLGym[2]

To use the tool, a user provides a YAML file that defines the variable(s) that need to be recorded for the multivariate signal, $w$, and the specification(s) that the signal needs to satisfy. Additionally, the user must provide the underlying Gym environment that will be augmented. Provided these two inputs, STLGym generates a new Gym environment where the specified variables are recorded so RTAMT can monitor the defined STL specification and return the robustness degree as the reward function.

### 4.1 Computing the Robustness Degree

To compute the robustness degree, we make use of RTAMT [19], a tool for monitoring STL specifications on recorded data. Given the recorded signal and specification, RTAMT computes the robustness degree according to the quantitative semantics described in Section 2.2. Whenever the robustness degree is calculated, it covers the full episode from time 0 to $t$.

### 4.2 Allowable Specifications

Our approach is amenable to a wide range of specifications and supports the full range of semantics described in Section 2.2 in addition to any described in

[2] STLGym implementation is available at `https://github.com/nphamilton/stl-gym`

RTAMT's readme[3]. This includes both timed and untimed operators, adding more options than allowed in a similar tool *Truncated Linear Temporal Logic* TLTL [14]. Furthermore, our approach allows for specifications to be broken up into individual parts. For example, consider the Cartpole example from Section 3.2. The desired behavior ("Keep the pole upright between $\pm 12^\circ$ and the cart within $\pm 2.4$ units") can be written as

$$\Phi_{single} = G((|\theta| < 0.20944) \wedge (|x| < 2.4)) \tag{3}$$

or it can be broken up into the individual components and combined with a conjunction,

$$\begin{aligned} \phi_{angle} &= G(|\theta| < 0.20944) \\ \phi_{position} &= G(|x| < 2.4) \\ \Phi_{split} &= \phi_{angle} \wedge \phi_{position}. \end{aligned} \tag{4}$$

These specifications, Equation 3 and Equation 4, are equivalent and allowable in both TLTL and STLGym. However, STLGym allows users to treat $\phi_{angle}$ and $\phi_{position}$ as individual specifications and automatically applies the conjunction. Any number of individual specifications can be defined, and the resulting specification the RL agent will learn to satisfy is the conjunction of all of them. Thus, if $n$ specifications are provided, the RL agent will learn to satisfy

$$\Phi = \bigwedge_{i=0}^{n} \phi_i. \tag{5}$$

### 4.3 Calculating Reward

STLGym replaces any existing reward function in the environment with the *robustness degree* calculated using the provided specification(s) and RTAMT. If the user defines $n$ specifications, $\phi_0, \phi_1, ..., \phi_n$ with corresponding weight values[4], $c_0, c_1, ..., c_n$, the reward function is constructed as

$$r_t = \sum_{i=0}^{n} c_i \rho(s, \phi_i, 0). \tag{6}$$

We include optional weights to add more versatility. This allows for users to write specifications that build on each other, i.e. a specification is defined using another specification, but remove one from the reward function if desired by setting its weight to 0. Additionally, weights can help establish priorities in learning specifications. For example, we go back to the CartPole specification Equation 4. The reward function generated, according to the quantitative semantics described in Section 2.2, for the specification is

$$r_t = c_{angle} \min_{t' \in [0,t]} (0.20944 - |\theta_{t'}|) + c_{position} \min_{t' \in [0,t]} (2.4 - |x_{t'}|). \tag{7}$$

[3] The RTAMT code is available at `https://github.com/nickovic/rtamt`

[4] If a weight is not defined by the user, the default is 1.

If both $c_{angle} = c_{position} = 1$, then the maximum possible reward for satisfying both specifications is 2.60944. However, because the environment was designed to terminate if either specification is violated, if the agent only satisfies $\phi_{position}$ and lets the pole fall, the maximum possible reward is 2.4. Since the gain from keeping the pole upright is so small, it could be ignored. In contrast, if we make the weights $c_{angle} = 4.7746$ and $c_{position} = 0.41666$, then the maximum possible reward for satisfying both specifications is 2. If either of the specifications are ignored, the maximum possible reward drops to 1. Thus, we have enforced equal priority for satisfying the specifications.

**Dense vs Sparse** In addition to adding optional weights for each specification, STLGym allows users to specify if the reward function should be calculated densely or sparsely. This design decision was spurred on by the existing RL literature, where there are two main types of rewards utilized: dense and sparse. In the literature, dense rewards are returned at every timestep and are often a scalar representation of the agent's progresses toward the goal. For example, the baseline reward function in the Pendulum environment (Equation 2) is a dense reward. In contrast, sparse rewards are not returned at each timestep, but instead are only returned if certain conditions are met. For example, an agent receiving +1 for passing a checkpoint would be considered a sparse reward. Each of these reward types have their advantages for different tasks and algorithms. However, we make use of these terms to make our own definitions of dense and sparse reward as they relate to frequency.

**Definition 1 (Dense Reward).** *When using the dense reward, the robustness degree is computed at every allowable timestep. Thus, at each timestep, the reward returned to the agent is the robustness degree of the episode from the beginning to the current time step.*

**Definition 2 (Sparse Reward).** *When using the sparse reward, the robustness degree is only computed once at the end of the episode. In all timesteps before that, the reward is 0. Thus, the return is the robustness degree for the entire episode.*

From our experiments, we found using dense rewards trained agents to satisfy the specification with fewer timesteps, while the sparse reward was better for evaluating their performance and understanding if they have successfully learned to satisfy the specification or not. An example is provided in Section 5.1.

## 5 Example Case Studies

In this section, we describe 5 case studies we conducted using the environments described in Section 3[5]. In all of our case studies, we use the Proximal Policy

[5] All training scripts are available at `https://github.com/nphamilton/spinningup/tree/master/spinup/examples/sefm2022`

Optimization (PPO) [22] algorithm for training, unless otherwise specified. These case studies were designed to highlight features of STLGym and try to identify some potential "best practices" for future use in other environments.

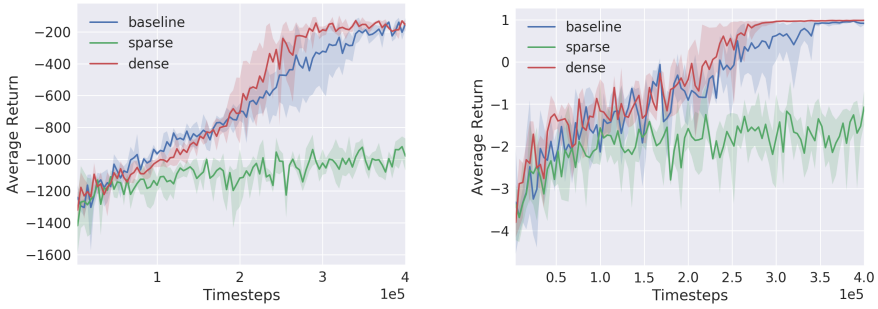
### 5.1 Sparse vs Dense Reward

In this case study, we demonstrate why having the ability to swap between sparse and dense versions of our STL reward function is important. To this end, we train 30 agents in the pendulum environment from Section 3.1 to swing the pendulum upright and stay upright. Written as an STL specification, that is

$$\Phi = F(G((|\theta| < 0.5))). \tag{8}$$

Ten agents are trained using the baseline reward function (Equation 2), ten agents are trained with the sparse version of our STL reward function, and ten agents are trained with the dense version of our STL reward function. Using the quantitative semantics from Section 2.2, our tool automatically generates the reward function,

$$r_t = \max_{t' \in [0,t]} \Big( \min_{t'' \in [t',t]} (0.5 - |\theta_{t''}|) \Big). \tag{9}$$




(a) Sample complexity of PPO agents trained in the Pendulum environment. The return is calculated using Equation 2.

(b) Sample complexity of PPO agents trained in the Pendulum environment. The return is calculated using Equation 9 defined sparsely.

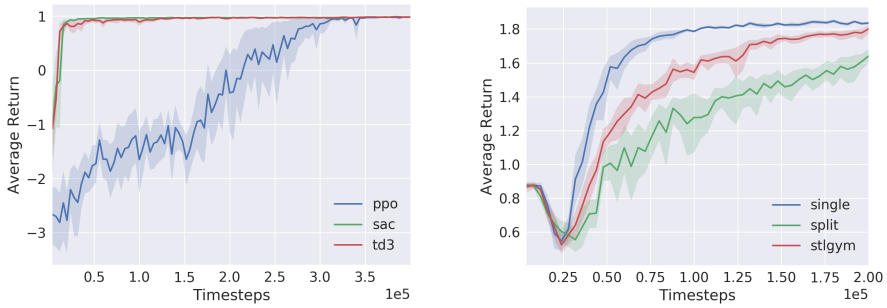
**Fig. 3.** Plots comparing the sample complexity using PPO to train agents in the Pendulum environment using three reward functions: (baseline) the baseline reward function, Equation 2; (sparse) the STLGym reward function, Equation 9, defined sparsely; and (dense) the STLGym reward function defined densely. Each curve represents the average return from 10 agents trained the same way. The shaded region around each curve shows the 95% confidence interval.

We show the sample complexity plots of training these 30 agents with the 3 different reward functions in Fig. 3. Sample complexity is a measure of how

quickly an RL agent learns optimal performance. Throughout training, the process is halted, and the agent is evaluated to see how well it performs with the policy learned so far. The policy is evaluated in ten episodes, and the performance, measured by the return, is recorded for the plot. A better sample complexity is shown by a higher return earlier in training. In Fig. 3, we show sample complexity measured by the (a) baseline reward function and (b) the sparse STL reward function to highlight how the agents trained with the dense STL reward have a better sample complexity than agents trained with the baseline reward function even according to the baseline metric.

While the agents trained using the sparse STL reward function failed to learn an optimal policy, using the sparse STL reward function for evaluating performance was very beneficial. Using the dense reward function for evaluating performance is very similar to the baseline reward function, in that neither provide any insight into whether or not the learned policy satisfies the desired behavior. In contrast, using the sparse STL reward function in Fig. 3(b), we see the exact point where the learned policies are successfully able to satisfy the specification when the return is greater than 0.

### 5.2 STLGym is Algorithm-Agnostic




(a) Comparing multiple RL algorithms using STLGym to learn the Pendulum specification, Equation 8.

(b) Comparing the three options presented in Section 5.3 in the CartPole.

**Fig. 4.** These plots compare the sample complexity of agents trained using different methods. Each curve represents the average of 10 trained agents, and the shaded region shows the 95% confidence interval. In (b), the return is calculated using the sparse definition of $\Phi_{split}$ (reward function represented by Equation 7) with $c_{angle} = 4.7746$ and $c_{position} = 0.41666$ so the maximum possible return is 2.0.

In this case study, we demonstrate that our approach is algorithm-agnostic by using multiple RL algorithms for the Pendulum example explained in Section 3.1. All algorithms are used to learn the optimal policy for satisfying the specification

in Equation 8. We demonstrate the following RL algorithms successfully learning to satisfy the specification using STLGym: Proximal Policy Optimization (PPO) [22], Soft Actor-Critic (SAC) [7], and Twin Delayed Deep Deterministic Policy Gradient (TD3) [6]. The sample complexity plot in Fig. 4(a) shows all RL algorithms successfully learn to satisfy the specification. While the results suggest SAC and TD3 work better with our STL reward function, these algorithms are known to learn the optimal policy for this environment very quickly. More examples, across different environments, are needed to make that claim.

### 5.3 On Separating Specifications and Scaling

The goal of the agent in the CartPole environment is to learn how to keep the pole upright so the angle, $\theta$, is between $\pm 12^\circ$ and the cart's position, $x$ remains within the boundary of $\pm 2.4$ for 200 timesteps. As explained in Section 4.2, this specification can be written as a singular specification, Equation 3, or as the conjunction of individual components, Equation 4.

Using STL's quantitative semantics, STLGym would generate the reward function for $\Phi_{single}$ as

$$r_t = \min_{t' \in [0,t]} \Big( \min \big( (0.20944 - |\theta_{t'}|), (2.4 - |x_{t'}|) \big) \Big). \tag{10}$$

Similarly, STLGym would generate the reward function for $\Phi_{split}$ as Equation 7

In this case study, we look at how splitting up the specification into its individual components creates a different reward function that impacts the training. We compare the sample complexity of learning $\Phi_{single}$ against learning $\Phi_{split}$ with and without weights. The results are shown in Fig. 4(b).

The results shown in Fig. 4(b) indicate splitting the specification is a hindrance for learning. The agents that were trained to satisfy $\Phi_{single}$ (single), converged to a more optimal policy faster than both the weighted (stlgym) and unweighted (split) options of $\Phi_{split}$. We expect this is a direct result of trying to satisfy $\Phi_{single}$, where the robustness degree is always the worst-case of satisfying both the angle and positions specifications. There is no credit awarded for satisfying one better than the other, like in the $\Phi_{split}$ definition. We believe that, while splitting the specification in this case study was more of a hindrance, in more complicated systems with more specifications, splitting could be more beneficial than shown here. In those cases, the option for weighting the individual specifications will be very helpful as the weighted and split option (stlgym), which is only supported in STLGym, learned faster than and outperformed the unweighted option.

### 5.4 Retraining With New Goal

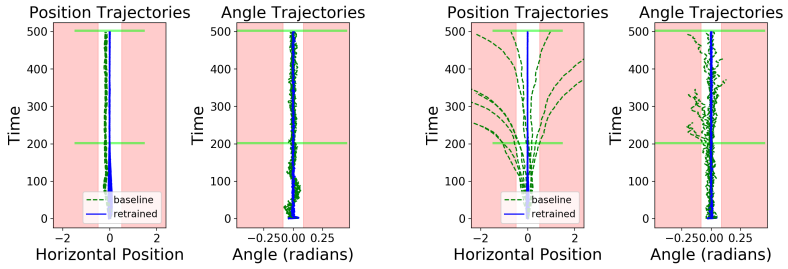
There are many cases where the traditional reward functions successfully train agents to complete the desired behavior, but we want to refine/improve/augment that behavior to some other desired behavior. Instead of designing a new reward

function and training a new agent from scratch, our tool can be leveraged to retrain the agent to satisfy the new desired behavior. This also makes our tool amenable to curriculum learning [26], an RL training strategy that trains agents in progressively harder environments or constraints. Similar to a learning curriculum used to teach students in a class, by starting with easier constraints and building upon what is learned from the easier tasks, the agent is better able to learn more complex behaviors.

In this case study, we look at an example with the CartPole environment described in Section 3.2. The baseline reward function trains agents to keep the pole upright very efficiently, but as [2] point out in their work, many of the learned policies are unstable. When they evaluated the policies for longer than 200 timesteps, they found many learned policies failed shortly after 200 timesteps. We saw similar results, which are shown in Fig. 5. To counteract this issue, we retrain the agents to maximize the measured robustness of the specifications

$$\begin{aligned} \phi_{position} &= F(G(|x| < 0.5)), \text{ and} \\ \phi_{angle} &= F(G(|\theta| < 0.0872665)). \end{aligned} \tag{11}$$

In plain English, the specifications translate to "eventually the cart will always be within $\pm 0.5$ units of the center of the track" and "eventually, the pole's angle will always be within $\pm 5°$."[6]

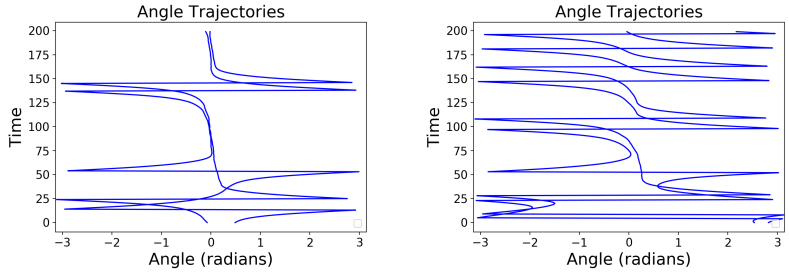



(a) 10 example episodes where the policy learned using the baseline reward function is stable.

(b) 10 example episodes where the policy learned using the baseline reward function is unstable.

**Fig. 5.** These plots show recorded episodes of trained policies evaluated in the CartPole environment. The red marks the region outside the specification and the horizontal green lines mark the goal during training at 200, and the goal at evaluation 500. In (a) we see the agent trained with the baseline reward function learned a stable policy and retraining with STLGym is able to further refine the learned policy to maximize the distance to the red region. In (b) we see the agent trained with the baseline reward learned an unstable policy, but after retraining with STLGym, the learned policy becomes stable.

[6] These specifications came from [2].




(a) Two recorded episodes of one trained policy.

(b) Two recorded episodes of a different trained policy.

**Fig. 6.** Episodes of policies trained to satisfy the timed specification in Equation 12.

After some retraining, Fig. 5 shows the retrained policies converged to more stable and consistent behavior. In particular, Fig. 5.b shows our approach corrects the unstable behavior.

### 5.5 Learning a Timed Specification

In this case study, we look at one of the features of our tool that sets it apart from almost all existing approaches in the literature—the ability to learn timed specifications. Here we return to the Pendulum environment described in Section 3.1. This time, the specification is "eventually the angle will be between $\pm 45°$ for 10 timesteps." In STL, the desired behavior is written as,

$$\Phi = F(G_{[0:10]}(|\theta| < 0.5)). \tag{12}$$

And is converted by our tool to the reward function,

$$r_t = \max_{t' \in [0,t]} \Big( \min_{t'' \in [t', t'+10]} (0.5 - |\theta_{t''}|) \Big). \tag{13}$$

The results of learning the specification in Equation 12 are highlighted in Fig. 6 where we show a few example episodes. When we first wrote this specification, we believed the resulting behavior would closely match that of the agents in Section 5.1. Instead, the learned policies were more varied. Some stay close to the upright position for longer than others, but they always return. We believe this is a result of the circular state space, which puts the agent back in a starting position after it moves away from upright. This result shows STLGym can successfully train agents to satisfy timed specifications. However, it also highlights a limitation of our approach: we have no way of overwriting the terminal conditions. We would see more consistent results if we were able to stop the episode once the specification was satisfied, but that is a feature left for future work.

## 6 Related Work

Our work is not the first to use temporal logic specifications to create reward functions. The previous works can be grouped into two categories, (1) quantitative semantics and (2) reward machines. We describe the related works in greater detail below and provide a general comparison of our approach with others in Table 1. The RL algorithms listed in Table 1 are the following: Augmented Random Search (ARS) [17], Deep Deterministic Policy Gradient (DDPG) [15], Deep Q-Learning (DQN) [18], Neural Fitted Q-iteration (NFQ) [21], Relative Entropy Policy Search (REPS) [20], Q-Learning (Q) [27], and Twin Delayed Deep Deterministic Policy Gradient (TD3) [6].

**Table 1.** A comparison of our tool to similar tools in the literature, separated by category, filled in to the best of our knowledge. × indicates the feature is not supported, ✓ indicates the feature is supported, and ? indicates it should be supported, but we cannot say so with confidence.

| Name | Env-API | Sparse/Dense | RL Algorithms | Retraining | Timed | Sequential |
|---|---|---|---|---|---|---|
| TLTL [14] | ? | Dense | REPS | ? | × | ✓ |
| BHNR [2] | Custom | Dense | DQN, PPO | ? | ✓ | ? |
| STLGym (ours) | Gym | Both | Any | ✓ | ✓ | ✓ |
| QRM [11] | Gym | Both | Q, DQN | × | × | ✓ |
| LCRL [10] | Custom | Both | Q, DDPG, NFQ | × | × | ✓ |
| SPECTRL [12] | Custom | Dense | ARS | × | × | ✓ |
| DIRL [13] | Gym | Dense | ARS, TD3 | × | × | ✓ |

### 6.1 Quantitative Semantics

The quantitative semantics category is where our work resides. These works, [1, 2, 14], generate reward functions based on the quantitative semantics of the temporal logics used to write the specifications the RL agents are tasked with learning to satisfy. In Truncated Linear Temporal Logic (TLTL), presented in [14], the authors create a new specification language, TLTL, that consciously removes the time bounds from STL to only have untimed operators. They made this decision, so specifications do not have to account for robotic limitations. In contrast, our STLGym is designed to handle both timed and untimed specifications, thus handling all TLTL problems and more.

Another work, [2], uses timed and untimed STL specifications similar to our STLGym. Their approach, Bounded Horizon Nominal Robustness (BHNR), computes a normalized robustness value over bounded horizons, i.e. small segments, of the episode, creating a reward vector. By only analyzing the robustness over smaller segments of the episode, their approach is able to speed up the robustness degree calculation for dense reward computation. However, because only a small portion of the episode is analyzed, their approach cannot be

used to determine the robustness degree across an entire episode like our sparse reward function is able to do. Additionally, their implementation limits user's specifications to be defined only by variables in the environment's observation space. Thus, their tool cannot train our pendulum example without re-writing to specification in terms of $x$ and $y$ instead of $\theta$.

### 6.2 Reward Machines

Reward machine approaches, [5, 11–13], use finite state automata (FSA) to handle context switching in the reward function. Temporal logic specifications are used to generate FSA that monitor the episode for satisfaction. Additionally, depending on which state of the FSA is in, the reward function changes in order to guide the agent towards satisfying the next specification. This approach is optimal for solving sequential tasks because it allows the user to specify "go to the fridge; open the door; take something out; close the door; return to home" and the reward function changes depending on which part of the task is being done. To the best of our knowledge, however, none of these approaches can handle timed specifications yet.

## 7 Conclusions and Future Work

This paper presents our tool, STLGym, for training agents to satisfy timed and untimed STL specifications using RL. To demonstrate the features of our tool and explore some best practices for learning to satisfy STL specifications, we trained over 130 different RL agents in our 5 case studies. From these case studies we observed (1) RL agents learned STLGym's dense rewards better than sparse rewards, (2) STLGym is algorithm-agnostic and works with any RL algorithm designed to integrate with Gym environments, (3) leaving specifications combined is better for RL agents than splitting them into individual parts, (4) STLGym is effective for retraining RL agents to better satisfy specifications, and (5) STLGym is effective for training RL agents to satisfy timed STL specifications.

In future work, we hope to expand to other, more complicated environments and explore more scenarios with timed specifications. Additionally, we would like to explore how STLGym can be leveraged more effectively for curriculum learning.

**Acknowledgments** The material presented in this paper is based upon work supported the Defense Advanced Research Projects Agency (DARPA) through contract number FA8750-18-C-0089, the Air Force Office of Scientific Research (AFOSR) award FA9550-22-1-0019, the National Science Foundation (NSF) through grant number 2028001, and the Department of Defense (DoD) through the National Defense Science & Engineering Graduate (NDSEG) Fellowship Program. Any opinions, findings, and conclusions or recommendations expressed in this publication are those of the authors and do not necessarily reflect the views of DARPA, AFOSR, NSF or DoD.

## References


1. Aksaray, D., Jones, A., Kong, Z., Schwager, M., Belta, C.: Q-learning for robust satisfaction of signal temporal logic specifications. In: 2016 IEEE 55th Conference on Decision and Control (CDC). pp. 6565–6570. IEEE (2016)
2. Balakrishnan, A., Deshmukh, J.V.: Structured reward shaping using signal temporal logic specifications. In: 2019 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS). pp. 3481–3486. IEEE (2019)
3. Barto, A.G., Sutton, R.S., Anderson, C.W.: Neuronlike adaptive elements that can solve difficult learning control problems. IEEE transactions on systems, man, and cybernetics **SMC-13**(5), 834–846 (1983)
4. Brockman, G., Cheung, V., Pettersson, L., Schneider, J., Schulman, J., Tang, J., Zaremba, W.: Openai gym (2016)
5. Camacho, A., Icarte, R.T., Klassen, T.Q., Valenzano, R.A., McIlraith, S.A.: Ltl and beyond: Formal languages for reward function specification in reinforcement learning. In: IJCAI. vol. 19, pp. 6065–6073 (2019)
6. Fujimoto, S., Hoof, H., Meger, D.: Addressing function approximation error in actor-critic methods. In: International Conference on Machine Learning. pp. 1587–1596. PMLR (2018)
7. Haarnoja, T., Zhou, A., Abbeel, P., Levine, S.: Soft actor-critic: Off-policy maximum entropy deep reinforcement learning with a stochastic actor. In: International Conference on Machine Learning. pp. 1861–1870. PMLR (2018)
8. Hamilton, N., Musau, P., Lopez, D.M., Johnson, T.T.: Zero-shot policy transfer in autonomous racing: reinforcement learning vs imitation learning. In: Proceedings of the 1st IEEE International Conference on Assured Autonomy (2022)
9. Hamilton, N., Schlemmer, L., Menart, C., Waddington, C., Jenkins, T., Johnson, T.T.: Sonic to knuckles: evaluations on transfer reinforcement learning. In: Unmanned Systems Technology XXII. vol. 11425, p. 114250J. International Society for Optics and Photonics (2020)
10. Hasanbeig, M., Abate, A., Kroening, D.: Logically-constrained reinforcement learning code repository. `https://github.com/grockious/lcrl` (2020)
11. Icarte, R.T., Klassen, T., Valenzano, R., McIlraith, S.: Using reward machines for high-level task specification and decomposition in reinforcement learning. In: International Conference on Machine Learning. pp. 2107–2116. PMLR (2018)
12. Jothimurugan, K., Alur, R., Bastani, O.: A composable specification language for reinforcement learning tasks. In: Advances in Neural Information Processing Systems 32: Annual Conference on Neural Information Processing Systems 2019, NeurIPS 2019 (2019)
13. Jothimurugan, K., Bastani, O., Alur, R.: Abstract value iteration for hierarchical reinforcement learning. In: International Conference on Artificial Intelligence and Statistics. pp. 1162–1170. PMLR (2021)
14. Li, X., Vasile, C.I., Belta, C.: Reinforcement learning with temporal logic rewards. In: 2017 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS). pp. 3834–3839. IEEE (2017)
15. Lillicrap, T.P., Hunt, J.J., Pritzel, A., Heess, N., Erez, T., Tassa, Y., Silver, D., Wierstra, D.: Continuous control with deep reinforcement learning. In: ICLR (2016)
16. Maler, O., Nickovic, D.: Monitoring temporal properties of continuous signals. In: Formal Techniques, Modelling and Analysis of Timed and Fault-Tolerant Systems, pp. 152–166. Springer (2004)

17. Mania, H., Guy, A., Recht, B.: Simple random search of static linear policies is competitive for reinforcement learning. In: Proceedings of the 32nd International Conference on Neural Information Processing Systems. pp. 1805–1814 (2018)
18. Mnih, V., Kavukcuoglu, K., Silver, D., Rusu, A.A., Veness, J., Bellemare, M.G., Graves, A., Riedmiller, M., Fidjeland, A.K., Ostrovski, G., et al.: Human-level control through deep reinforcement learning. nature **518**(7540), 529–533 (2015)
19. Ničković, D., Yamaguchi, T.: Rtamt: Online robustness monitors from stl. In: International Symposium on Automated Technology for Verification and Analysis. pp. 564–571. Springer (2020)
20. Peters, J., Mulling, K., Altun, Y.: Relative entropy policy search. In: Twenty-Fourth AAAI Conference on Artificial Intelligence (2010)
21. Riedmiller, M.: Neural fitted q iteration–first experiences with a data efficient neural reinforcement learning method. In: European conference on machine learning. pp. 317–328. Springer (2005)
22. Schulman, J., Wolski, F., Dhariwal, P., Radford, A., Klimov, O.: Proximal policy optimization algorithms. arXiv preprint arXiv:1707.06347 (2017)
23. Silver, D., Huang, A., Maddison, C., Guez, A., Sifre, L., Driessche, G., Schrittwieser, J., Antonoglou, I., Panneershelvam, V., Lanctot, M., Dieleman, S., Grewe, D., Nham, J., Kalchbrenner, N., Sutskever, I., Lillicrap, T., Leach, M., Kavukcuoglu, K., Graepel, T., Hassabis, D.: Mastering the game of go with deep neural networks and tree search. Nature **529**, 484–489 (Jan 2016)
24. Silver, D., Singh, S., Precup, D., Sutton, R.S.: Reward is enough. Artificial Intelligence **299**, 103535 (2021)
25. Vinyals, O., Babuschkin, I., Czarnecki, W.M., Mathieu, M., Dudzik, A., Chung, J., Choi, D.H., Powell, R., Ewalds, T., Georgiev, P., et al.: Grandmaster level in starcraft ii using multi-agent reinforcement learning. Nature **575**(7782), 350–354 (2019)
26. Wang, X., Chen, Y., Zhu, W.: A survey on curriculum learning. IEEE Transactions on Pattern Analysis and Machine Intelligence (2021)
27. Watkins, C.J., Dayan, P.: Q-learning. Machine learning **8**(3), 279–292 (1992)
28. Wurman, P.R., Barrett, S., Kawamoto, K., MacGlashan, J., Subramanian, K., Walsh, T.J., Capobianco, R., Devlic, A., Eckert, F., Fuchs, F., et al.: Outracing champion gran turismo drivers with deep reinforcement learning. Nature **602**(7896), 223–228 (2022)