Combining Neural Networks and Tree Search for Task and Motion Planning in Challenging Environments

Chris Paxton[1,2], Vasumathi Raman[1], Gregory D. Hager[2], Marin Kobilarov[1,2]

***Abstract*—We consider task and motion planning in complex dynamic environments for problems expressed in terms of a set of Linear Temporal Logic (LTL) constraints, and a reward function. We propose a methodology based on reinforcement learning that employs deep neural networks to learn low-level control policies as well as task-level option policies. A major challenge in this setting, both for neural network approaches and classical planning, is the need to explore future worlds of a complex and interactive environment. To this end, we integrate Monte Carlo Tree Search with hierarchical neural net control policies trained on expressive LTL specifications. This paper investigates the ability of neural networks to learn both LTL constraints and control policies in order to generate task plans in complex environments. We demonstrate our approach in a simulated autonomous driving setting, where a vehicle must drive down a road in traffic, avoid collisions, and navigate an intersection, all while obeying given rules of the road.**

I. Introduction

A robot operating in the physical world must reason in a hybrid space: both its continuous motion in the physical world and the discrete goals it must accomplish are pertinent to correctly completing a complex task. Common practice is to first compute a discrete action plan, then instantiate it depending on what is physically feasible. The field of Task and Motion Planning (TAMP) seeks to integrate the solving of the continuous and discrete problems since a robot's immediate physical motion is inextricably coupled with the discrete goal it seeks.

One particular case where TAMP is particularly relevant is in the domain of autonomous driving. Self-driving cars have to deal with a highly complex and dynamic environment: they share a road with other moving vehicles, as well as with pedestrians and bicyclists. Road conditions are also unpredictable, meaning that such methods must be capable of dealing with uncertainty.

Current TAMP approaches combine high-level logical planning with continuous space motion planning. These methods succeed at solving many sequential path planning and spatial reasoning problems [20, 22], but dealing with dynamic environments and complex constraints is still a challenge [16]. The problem is that the combined discrete and continuous state space tends to explode in size for complex problems. The addition of temporal constraints makes the search problem even more difficult, though there has been recent progress in this direction [16].

On the other hand, recent work in Deep Reinforcement Learning (DRL) has shown promise in control policy learning in challenging domains [13, 21, 15, 4], including robotic manipulation [9] and autonomous driving [4, 23, 19]. DRL has also been combined with Monte Carlo Tree Search (MCTS) for learning [11] and game playing [21] in a variety of discrete and continuous environments. However, the question remains open whether these approaches can be integrated in a TAMP framework to produce reliable robot behavior.

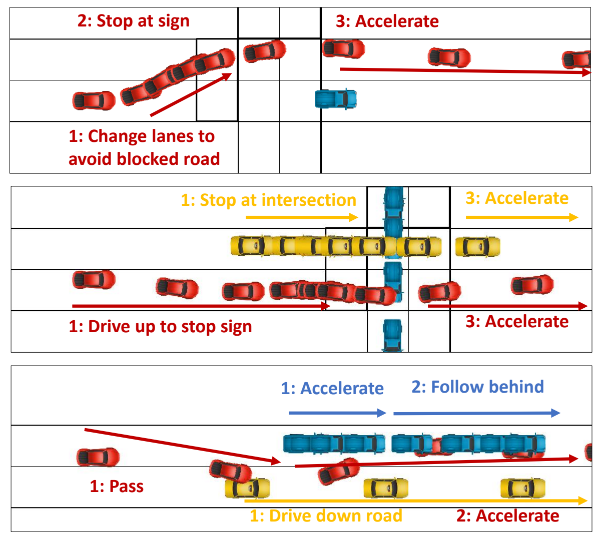

Figure 1: Simulated self driving car problems containing an intersection and multiple vehicles. The car must be able to behave intelligently when confronted with sudden obstacles like stopped or slow moving vehicles, and it must be able to obey the rules of the road.

In this paper, we show that the best of both worlds can be achieved by using neural networks to learn both low-level control policies and high-level action selection priors, and then using these multi-level policies as part of a heuristic search algorithm to achieve a complex task. A major challenge of complex, dynamic environments is that there is no straightforward heuristic to guide the search towards an optimal goal: since the goals are temporally specified, there is no way to anticipate constraints and conflicts that may arise further on in the plan. To address this issue, we formulate task and motion planning as a variant of Monte Carlo Tree Search over high-level options, each of which is represented by a learned control policy, trained on a set of LTL formulae. This approach allows us to efficiently explore the relevant parts of the search space to find high quality solutions when other methods would fail to do so.

To summarize, the contributions of this paper are:

- A planning algorithm combining learned low-level control policies with learned high level "option policies" over these low-level policies for TAMP in dynamic environments.
- A framework for incorporating complex task requirements expressed in temporal logic
- Evaluation of our approach in a simulated autonomous driving domain.

Limitations are discussed in Section VI.

II. PRELIMINARIES

In this section we establish notation and terminology for the system modeling and task specification formalisms, Markov Decision Processes (MDPs) and linear temporal logic (LTL), respectively.

A. System model

In keeping with most reinforcement learning algorithms, we model the system under consideration as a Markov Decision Process (MDP) [2]. Learning is performed over a sequence of time steps. At each step t, the agent observes a state, $s_t \in S$, which represents the sensed state of the system, i.e., its internal state as well as what it perceives about the environment it operates in. Based on s_t, the agent selects an action $a_t \in A$ from an available set of actions. On performing a_t in s_t, the agent receives an immediate reward, $r_t \in R$, and moves to a state in set $s_{t+1} \in \delta(s_t, a_t)$. The goal of the agent is to maximize its cumulative reward or a time-discounted sum of rewards over a time horizon (which may be finite or infinite). Without loss of generality, the agent acts according to a policy, $\pi : S \to A$. A *run* of an MDP $\mathbf{s} = s_0 s_1 s_2 \cdots$ is an infinite sequence of states such that for all t, exists $a_t \in A$ such that $s_{t+1} \in \delta(s_t, a_t)$.

Given a current state s, the value of $Q^\pi(s, a)$ is defined to be the best cumulative reward that can be obtained in the future under policy π after performing action a. The Q-function is thus a local measure of the quality of action a. Similarly, the *value function* of an MDP $V^\pi : S \to R$ is a local measure of the quality of s under policy π. For an optimal policy π^*, V^* and Q^* are obtained as fixed points using Bellman's equation. Most reinforcement learning algorithms approximate either the the V function or the Q function. For more detail, the interested reader is referred to the survey of RL methods by [12].

To solve the MDP, we pick from a hypothesis class of policies π composed using a set of high-level options, which are themselves learned from a hypothesis class of parametrized control policies using a deep neural network. As such, the optimal policy may not be contained in this hypothesis class, but we are able to demonstrate architectures under which a good approximation is obtained.

B. Linear Temporal Logic

We prescribe properties of plans in terms of a set of atomic statements, or propositions. An atomic proposition is a statement about the world that is either `True` or `False`. Let AP be a finite set of atomic propositions, indicating properties such as occupancy of a spatial region, and a labeling function $\mathcal{L} : S \to 2^{AP}$ a map from system states to subsets of atomic propositions that are `True` (the rest being false). For a given run $\mathbf{s}$, a *word* is the corresponding sequence of labels $\mathcal{L}(\mathbf{s}) = \mathcal{L}(s_0)\mathcal{L}(s_1)\mathcal{L}(s_2)\cdots$. We use linear temporal logic (LTL) to concisely and precisely specify permitted and prohibited system behaviors in terms of the corresponding words. To allow users who may be unfamiliar with LTL to define specifications, some approaches such as that of [18] include a parser that automatically translates English sentences into LTL formulas. Many applications distinguish two primary types of properties allowed in a specification – *safety* properties, which guarantee that "something bad never happens", and *liveness* conditions, which state that "something good (eventually) happens". These correspond naturally to LTL formulas with operators "always" ($\Box$) and "eventually" ($\Diamond$).

One useful property of LTL is the equivalence between LTL formulas and Deterministic Rabin Automata (DRAs). Any LTL formula φ over variables in AP can be automatically translated into a corresponding DRA $\mathcal{A}_\varphi$ of size automaton $2^{2^{|AP|}}$ that accepts all and only those words that satisfy φ [6]. We refer the reader to [6] for details on LTL.

III. APPROACH

We consider systems that evolve according to continuous dynamics f_c and discrete dynamics f_d:

$$x' = f_c(x, w, u, o), \;\; w' = f_d(x, w, u, o)$$

where

- $x \in \mathcal{X} \subseteq \mathbb{R}^n$ is the continuous state
- $u \in \mathcal{U} \subseteq \mathbb{R}^m$ is the continuous control input
- $w \in \mathcal{W}$ is the discrete (logical) world state
- $o \in \mathcal{O}$ is a discrete (logical) option from a finite set $\mathcal{O}$

Our atomic propositions $p \in AP$ are defined as functions over the discrete world state, i.e. $p : \mathcal{W} \to \{\texttt{True}, \texttt{False}\}$.

In the MDP framework, $S = \mathcal{X} \times \mathcal{W}, A = \mathcal{U} \times \mathcal{O}, \delta(xw, uo) = x'w'$ such that $x' = f_c(x, w, u, o), \;\; w' = f_d(x, w, u, o)$. The labeling function over states is $\mathcal{L}(xw) = \{p \in AP \text{ such that } p(w) = \texttt{True}\}$.

We decompose the system into many actors. Each independent entity is an actor, and in particular the agent under control is an actor. A world state $s = xw \in \mathcal{X} \times \mathcal{W}$ consists of an environment $e \in \mathcal{E}$ and some number of actors N. The i-th world state in a sequence is therefore fully defined as:

$$x_i w_i = \langle x_{0,i} w_{0,i}, x_{1,i} w_{1,i}, \ldots, x_{N,i} w_{N,i}, e \rangle \, ,$$

where each actor k's state $x_{k,i} \in \mathbb{R}^n$ and $w_{k,i} \in \mathcal{W}_k$ such that $\sum_k n_k = n$ and $\prod \mathcal{W}_k = \mathcal{W}$, and actor 0 is the planner. Actors represent other entities in the world that will update over time according to some unspecified policy specific to them.

Finally, we assume we are given a feature function $\phi : S \to \mathcal{F}$, which computes a low-dimensional representation of the world state containing all information needed to compute a policy. For example, when doing end-to-end visuomotor learning as in [4, 23], $\phi(s)$ would simply return the camera image associated with that particular world state. We show other examples of this feature function in our experiments.

We seek to learn a set of behaviors that allow an actor to plan safe, near-optimal trajectories through the environment out to a fixed time horizon while obeying a set of discrete constraints. We decompose this problem into finding two sets of policies: a policy $\pi_{\mathcal{O}} : \mathcal{F} \to \mathcal{O}$ over high-level actions and a policy $\pi_{\mathcal{U}} : \mathcal{O} \times \mathcal{F} \to \mathcal{U}$ over low-level controls, such that their composition solves the MDP. In particular, our first subgoal is to compute a policy $\pi^*_{\mathcal{U}}(\cdot, o)$ for each high-level option o that maps from arbitrary feature values to controls:

$$\pi^*_{\mathcal{U}}(\phi(xw), o) = \arg\max_u (V^*(\delta(xw, uo)))$$

We also compute a second policy over options, $\pi^*_{\mathcal{O}}$:

$$\pi^*_{\mathcal{O}}(\phi(xw)) = \arg\max_o (V^*(\delta(xw, \pi^*_{\mathcal{U}}(\phi(xw), o)o)))$$

This high-level policy tells us what we expect to be the best control policy to execute over some short time horizon. Note that since we are imposing additional structure on the final policy (which takes the form $\pi^*(s) = \pi^*_{\mathcal{U}}(\phi(s), \pi^*_{\mathcal{O}}(\phi(s)))$), it may no longer be truly optimal. However, it is the optimal such policy found with this set of options $\mathcal{O}$. Our results show that decomposing the problem in this way, by learning a simple set of options to plan with, is effective in the autonomous driving domain. Without leveraging the inherent structure of the domain in this manner, end-to-end training would learn $\pi^*(s)$ directly, but require a much more sophisticated training method and a lot more data. Fig. 2 provides a schematic overview of our proposed approach.

A. Planning Algorithm

In a dynamic environment with many actors and temporal constraints, decomposing the problem into reasoning over goals and trajectories separately as in prior work [22, 20] is infeasible. Instead, we use learned policies together with an approach based on MCTS. The most commonly used version of MCTS is the Upper Confidence Bound (UCB) for Trees. We recursively descend through the tree, starting with $s = s_0$ as the current state. At each branch we would choose a high level option according to the UCB metric:

$$Q(s_i, o_i) = Q^*(s_i, o_i) + C\sqrt{\frac{\log(N(s_i, o_i))}{N(s_i) + 1}}$$

where $Q^*(s_i, o_i)$ is the average value of option o_i from simulated play, $N(s_i, o_i)$ is the number of times option o_i was observed from s_i, and $N(s_i)$ is the number of times s_i has been visited during the tree search. C is an experimentally-determined, domain-specific constant.

Our particular variant of MCTS has two specializations. First, we replace the usual Upper Confidence Bound (UCB) weight with the term from [21] as follows:

$$Q(s_i, o_i) = Q^*(s_i, o_i) + C\frac{P(s_i, o_i)}{1 + N(s_i, o_i)}$$

where $P(s_i, o_i)$ is the predicted value of option o_i from state s_i. The goal of this term is to encourage exploration while focusing on option choices that performed well according to previous experience; it grants a high weight to any terms that have a high prior probability from our learned model.

Next, we use Progressive Widening to determine when to add a new node. This is a common approach for dealing with Monte Carlo tree search in a large search space [7, 24] that limits the maximum number of children of a given node to some sub-linear function of the number of times a world state has been visited, commonly implemented as

$$N_{children}(s_i)^* = (N(s_i))^\alpha$$

with $\alpha \in (\frac{1}{2}, \frac{1}{4})$. Whenever we add a new node to the search tree, we use the current high-level policy to explore until we reach a termination condition.

After selecting an option to explore, we call the SIMULATE function to evolve the world forward in time. During this update, we check the full set of LTL constraints Φ and associated option constraints φ_o. If these are not satisfied, the search has arrived in a terminal node and a penalty is applied. Alg. 1 is the standard algorithm for Monte Carlo Tree Search with these modifications.

B. Model Checking

Each discrete option is associated with an LTL formula φ_o which establishes conditions that must hold while applying that option. We can evaluate $u_i = \pi_{\mathcal{U}}(o, \phi(x_i w_i))$ to get the next control as long as φ_o holds. In addition, we have a shared set Φ of LTL formulae that constrain the entire planning problem. In order to evaluate the cost function when learning options, as well as during MCTS, we check whether sampled runs satisfy an LTL formula. Since we are checking satisfaction over finite runs, we use a bounded-time semantics for LTL [3]. We precompute maximal accepting and rejecting strongly connected components of the DRA, which enables model checking in time linear in the length of the run.

IV. Self Driving Car Domain

We apply our approach to the problem of planning for a self-driving car passing through an all-way stop intersection. To successfully complete the task the car must: (1) accelerate to a reference speed, (2) stop at a stop sign, (3) wait until its turn to move, (4) accelerate back to the reference speed, all while avoiding collisions and changing lanes as necessary. We break this down into a set of mid-level options:

$$\mathcal{O} = \{\texttt{Default}, \texttt{Follow}, \texttt{Pass}, \texttt{Stop}, \texttt{Wait}, \texttt{Left}, \texttt{Right}, \texttt{Finish}\}$$

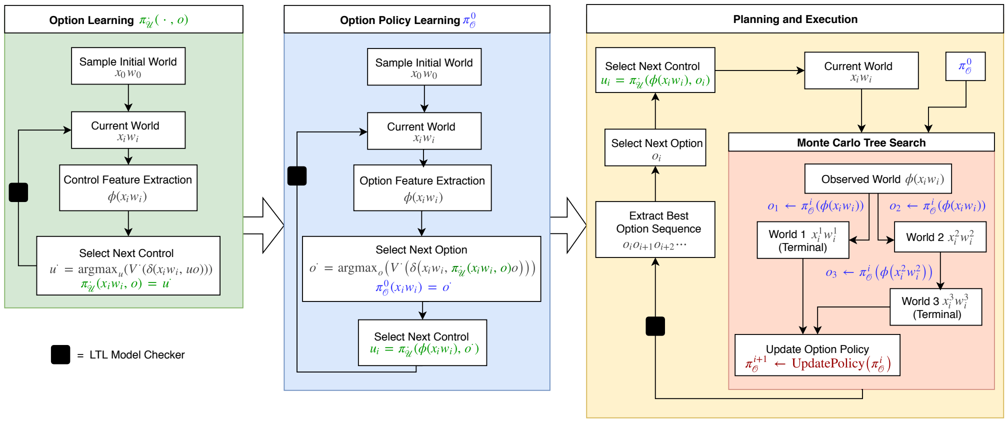

Figure 2: Workflow diagram of the proposed approach, showing the training process and planning/execution loop.

Algorithm 1 Pseudocode for MCTS over options.

function SELECT(s)
  if s is terminal **then return** $v(s)$
  else
    if $N(s) = 0$ or $N_{children}(s) < \sqrt{N(s)}$ **then**
      Add new edge with option o to children
    end if
    $o^* = \arg\max_o$
    if $N(s, o^*) = 0$ **then**
      // This branch has not yet been explored
      $s' =$ SIMULATE(s,o^*)
    else
      // This branch has been previously simulated
      $s' =$ LOOKUP(s,o^*)
    end if
    // Recursively explore the tree
    $v' =$ SELECT(s')
    // Update value associated with this node
    $v = v(s) + v'$
    UPDATE(v)
    return v
  end if
end function
function SIMULATE(s,o)
  while $t < t_{max}$ and $c(o)$ **do**
    $u^* = \pi^*_{\mathcal{U}}(\phi(s), o)$
    $s' =$ADVANCE_WORLD(s, u^*, Δt)
    $t = t + \Delta t$
    for $\varphi \in \Phi \cup \varphi_o$ **do**
      if s $\not\models \varphi$ **then**
        $s' \leftarrow terminal$
        return s'
      end if
    end for
  end while
  return s'
end function

where, for example, the `Default` option represents driving down a straight road and stopping at a stop sign. We employ a non-slip second-order nonholonomic model that is often sufficient for realistic motion planning in nominal driving conditions. The physical state of each vehicle entity in the world is defined by $x = (p_x, p_y, \theta, v, \psi)$, where (p_x, p_y) denotes the inertial position at the rear axle, θ is its heading relative to the road, v is the velocity and ψ is the steering angle. The control inputs are the acceleration a and steering angle rate $\dot{\psi}$, i.e. $u = (u_1, u_2) := (a, \dot{\psi})$. The dynamics of all vehicles are defined as

$$\dot{p}_x = v\cos\theta, \dot{p}_y = v\sin\theta,$$
$$\dot{\theta} = v\frac{\tan\psi}{L},$$
$$\dot{v} = u_1, \dot{\psi} = u_2,$$

where L is the vehicle wheel-base. These equations are integrated forward in time using a discrete-time integration using a fixed time-step $\Delta t = 0.1$ seconds during learning and for all experiments.

Our scenarios take place at the intersection of two two-lane, one-way roads. We choose this setting specifically because it creates a number of novel situations that the self-driving vehicle may need to deal with. Each lane is 3 meters wide with a 25 mph speed limit, corresponding to common urban or suburban driving conditions. The target area is a 90 meter long section of road containing an intersection, where two multi-lane one-way roads pass through each other. Stop signs are described as “stop regions”: areas on the road that vehicles must come to a stop in before proceeding.

Other vehicles follow an aggressive driving policy. If far enough away from an event, they will accelerate up to the speed limit (25 mph). Otherwise, they will respond accordingly. If the next event on the road is a vehicle, they will slow down to maintain a follow distance of 6 meters bumper to bumper. They will decelerate smoothly to come to a stop halfway through a stop region. They will remain stopped until the intersection is clear and they have priority over other waiting vehicles. Once in an intersection, they will accelerate up to the speed limit until they are clear. They have a preferred acceleration of $1.0m/s^2$, and will not brake harder than $2.0m/s^2$. Actors will appear on the road moving at either the speed limit or a point on the reference velocity curve (if before a stop sign).

A. Cost and Constraints

As described in Section III-B, each discrete option is associated with an LTL formula φ_o which establishes conditions that must hold while applying it, and we also have a shared set Φ of LTL formulae constraining the entire plan. For the road scenario, $\Phi =$

$$\square\,(\text{in\_stop\_region} \Rightarrow (\text{in\_stop\_region}\ \mathcal{U}\ \text{has\_stopped\_in\_stop\_region}))$$
$$\square((\text{in\_intersection} \Rightarrow \text{intersection\_is\_clear}) \wedge \neg\text{in\_intersection}\ \mathcal{U}\ \text{higher\_priority})$$

The reward function is a combination of a a cost term based on the current continuous state and a bonus based on completing intermediate goals or violating constraints (e.g. being rejected by the DRA corresponding to an LTL formula). The cost term penalizes the control inputs, acceleration and steering angle rate, as well as jerk, steering angle acceleration, and lateral acceleration. There are additional penalties for being far from the current reference speed and for offset from the center of the road. We add an additional penalty for being over the reference speed, set to discourage dangerous driving while allowing some exploration below the speed when interacting with other vehicles. This cost term is:

$$r_{cost} = \|(e_y, e_\theta, v - v_{ref}, \min(0, v_{ref} - v), a, \dot{a}, \dot{\psi})\|_W^2,$$

expressed as the weighted L_2 norm of the residual vector with respect to a diagonal weight matrix W. Here the errors e_y and e_θ encode the lateral error and heading error from the lane centerline.

We add terminal penalties for hitting obstacles or violating constraints. The other portion of the reward function are constraint violations. We set the penalty for constraint violations to -200 and provide a 200 reward for achieving goals: stopping at the stop sign and exiting the region. When training specific options, we set an additional terminal goal conditions when appropriate.

B. Learning

Our approach for multi-level policy learning is is similar to the framework used by [1], who use a multi-level policy that switches between multiple high-level options. This allows us to use state of the art methods such as [13, 10] for learning of individual continuous option-level policies. All control policies are represented as multilayer perceptrons with a single hidden layer of 32 fully connected neurons. We used the ReLu activation function on both the input and hidden layer, and the tanh activation function on the outputs. Outputs mapped to steering angle rate $\dot{\psi} \in [-1, 1]$ rad/s and acceleration $a \in [-2, 2]$ m/s^2.

These models were trained using Keras [5] and TensorFlow. We used the Keras-RL implementations of deep reinforcement learning algorithms DQN, DDPG, and continuous DQN [17]. These were trained for 1 million iterations with the Adam optimizer, a learning rate of $1e-3$. Exploration noise was generated via an Ornstein-Uhlbleck process with sigma that annealed between 0.3 and 0.1 over 500,000 iterations. We examined two different reinforcement learning approaches for finding the values of our low-level controllers: Deep Direct Policy Gradients (DDPG), as per [13], and Continuous Deep Q Learning (cDQN) [10]. However, cDQN had issues with convergence for some of our options, and therefore we only report results using low-level policies learned via DDPG.

We then performed Deep Q learning [14] on the discrete set of options to learn our high-level options policy. High-level policies were trained on a challenging road environment with 0 to 6 randomly placed cars with random velocity, plus a 50% chance of a stopped vehicle ahead in the current lane. We include features that capture the planning actor's heading and velocity on the road, as well as its relationship to certain other actors. We designed our set of features to be generalizable to situations with many or few other actors on the road. There is a fixed maximum horizon distance for distances to other entities in the world, with the maximum response at 0 distance and no response beyond this horizon. The feature set for the current actor based on its continuous state is:

$$\phi_x(s) = \left[v, v_{ref}, e_y, \psi, v\frac{\tan\psi}{L}, \theta, lane, e_{y,lane}, a, \dot{\psi}\right],$$

where we also included the previous control input $u = (a, \dot{\psi})$ (for clarity of presentation u is not explicitly included in the current state s). To this, we append the set of predicates:

$$\begin{aligned}\phi_w(s) = [&\texttt{not\_in\_stop\_region},\\ &\texttt{has\_entered\_stop\_region},\\ &\texttt{has\_stopped\_in\_stop\_region},\\ &\texttt{in\_intersection}, \texttt{over\_speed\_limit},\\ &\texttt{on\_route}, \texttt{intersection\_is\_clear},\\ &\texttt{higher\_priority}]\end{aligned}$$

For the vehicles ahead, behind, in the other lane ahead and behind, and on the cross road to the left and right, we add the set of features related to that actor (assuming its index is j, while the index of the vehicle for which the policy is applied to be i):

$$\phi_j(s_i) = [x_i - x_j, y_i - y_j, v_j, a_j, \texttt{waited}_j],$$

where `waited` is the accumulated time since the actor stopped at the stop sign. We also append the predicates associated with each actor. The complete feature set for vehicle i is thus $\phi = (\phi_x, \phi_w, \phi_{\{-i\}})$, where the notation $\{-i\}$ should be understood as the set of all indices but i.

V. Results

To validate the planning approach, we generated 100 random worlds in a new environment. Each world contains 0-5 other vehicles. These environments include more vehicles than training environments for individual options, and have the possibility of having many vehicles in the same lane as the learner. In addition, we test in a case where there are 0-5 random vehicles, plus one vehicle stopped somewhere in the lane ahead. This potentially requires merging into traffic in the other lane in order to proceed. In all cases, the vehicle needs

Table I: Comparison of different algorithms on 100 randomly-generated road environments with multiple other vehicles and with a stopped car on the road ahead.

Environment	Low-Level	High-Level	Constraint Violations	Collisions	Total Failures	Avg Reward	Std Dev Reward
No Stopped Car	Manual Policy	None	0	0	0	117.8	60.4
	256 NN Policy	None	3	4	6	109.4	95.7
	Learned	Manual	0	4	4	124.2	97.8
	Learned	Learned	0	0	0	137.8	74.2
Stopped Car Ahead	Manual Policy	None	0	19	19	27.2	142.1
	256 NN Policy	None	4	29	33	-9.1	149.3
	Learned	Manual	1	9	11	83.7	102.2
	Learned	Learned	0	3	3	95.2	74.2

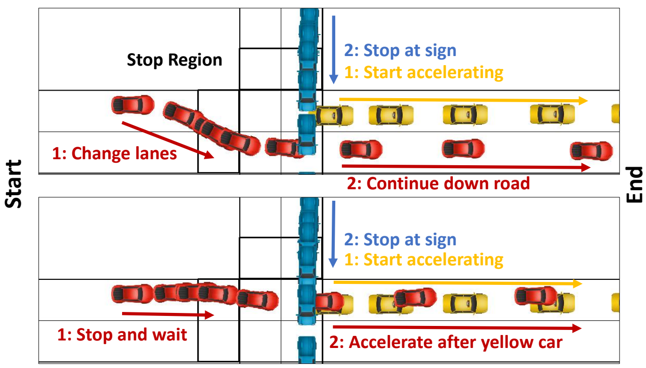

Figure 3: Two different solutions to safely passing through an intersection. The red car is using the proposed algorithm. Top: manual high-level policy changes lanes, bottom: learned high-level policy waits for car in front.

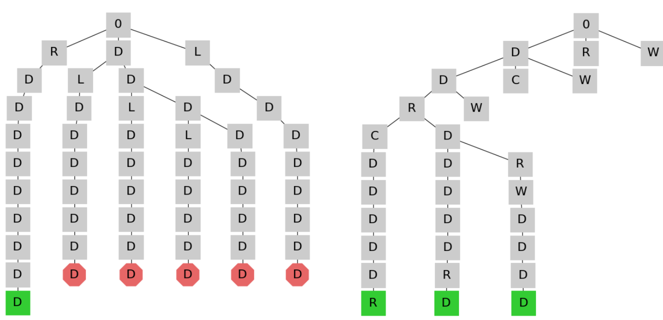

Figure 4: Comparison of MCTS on a test problem with a stopped car. Letters indicate option being executed: 0 = root, D = default "stay in lane" policy, W = wait, C = finish level, R = lane change right. On the left is tree search with a manual prior over options; the right tree uses the learned high-level policy. Green leaves indicate success; red leaves indicate failure.

to be able to negotiate an intersection despite the presence of other vehicles. We compare these two cases in Table I.

For cases with the learned or simple action sets, we performed 100 iterations of MCTS as per Alg. 1 to a depth of 10 seconds and select the best path to execute. Fig. 3 and Fig. 4 show how this functions in practice: the algorithm selects between its library of learned options, choosing to take an action that will move into a lane of moving traffic instead of into a lane where there is a stopped vehicle.

The "manual" policy in Table I executes the same aggressive policy as all the other actors on the road. It will usually come to a safe stop behind a stopped vehicle. The version of the planning algorithm with learned options but without the learned policy over options records a few collisions even in relatively simple problem with no stopped car. These situations occur in a few different situations, usually when the vehicle is "boxed in" by vehicles on multiple sides and needs to use a less-likely option to escape. Without the high-level policy to guide exploration, it does not know the correct sequence of options to avoid collisions.

By contrast, with the learned high-level policy, we see perfect performance on the test set for simple problems and three failures in complex problems. Note that these failures universally represent cases where the vehicle was trapped: there is a car moving at the same speed in the adjacent lane and a stopped car ahead. Given this situation, there is no good option other than to hit brakes as hard as possible. When our system does not start in such a challenging position, it will avoid getting into such a situation; if it predicts that such a system will arise in the next ten seconds, our planner would give us roughly 2 seconds of warning to execute an emergency stop and avoid collision. Fig. 4 shows a comparison of a subset of the planner calls for the version with the manually defined vs. learned high level policy, and Fig. 3 shows the selected trajectories associated with these trees.

VI. Conclusions

We laid out a framework for using learned models of skills to generate task and motion plans while making minimal assumptions as to our ability to collect data and provide structure to the planning problem. Our approach allows simple, off-the-shelf Deep Reinforcement Learning techniques to generalize to challenging new environments, and allows us to verify their behavior in these environments.

There are several avenues for improvement. First, the low-level policies learned in the self-driving car example could be improved. Second, more complex stermination conditions are possible under the proposed framework. Additionally, choosing the set of options is still done manually, and choosing the best set is still an open problem. Finally, in our examples we use a manually-determined feature set. In the future, we wish to extend this work to use stochastic control policies to perform a more inclusive search over possible trajectories. We will also apply our system to real robots.

References

[1] Jacob Andreas, Dan Klein, and Sergey Levine. Modular multitask reinforcement learning with policy sketches. *arXiv preprint arXiv:1611.01796*, 2016.

[2] Richard Ernest Bellman. *Introduction to the mathematical theory of control processes / Richard Bellman*. Academic Press New York, 1967. ISBN 0120848023.

[3] Armin Biere, Keijo Heljanko, Tommi A. Junttila, Timo Latvala, and Viktor Schuppan. Linear encodings of bounded LTL model checking. *Logical Methods in Computer Science*, 2(5), 2006. doi: 10.2168/LMCS-2(5:5)2006. URL http://dx.doi.org/10.2168/LMCS-2(5:5)2006.

[4] Mariusz Bojarski, Davide Del Testa, Daniel Dworakowski, Bernhard Firner, Beat Flepp, Prasoon Goyal, Lawrence D Jackel, Mathew Monfort, Urs Muller, Jiakai Zhang, et al. End to end learning for self-driving cars. *arXiv preprint arXiv:1604.07316*, 2016.

[5] François Chollet. Keras. https://github.com/fchollet/keras, 2015.

[6] Edmund M. Clarke, Orna Grumberg, and Doron Peled. *Model checking*. MIT Press, 2001. ISBN 978-0-262-03270-4.

[7] Adrien Couetoux, Mario Milone, Matyas Brendel, Hassen Doghmen, Michele Sebag, Olivier Teytaud, et al. Continuous rapid action value estimates. In *The 3rd Asian Conference on Machine Learning (ACML2011)*, volume 20, pages 19–31. JMLR, 2011.

[8] Xu Chu Ding, Stephen L. Smith, Calin Belta, and Daniela Rus. Optimal control of markov decision processes with linear temporal logic constraints. *IEEE Trans. Automat. Contr.*, 59(5):1244–1257, 2014. doi: 10.1109/TAC.2014.2298143. URL http://dx.doi.org/10.1109/TAC.2014.2298143.

[9] Ahmad El Sallab, Mohammed Abdou, Etienne Perot, and Senthil Yogamani. Deep reinforcement learning framework for autonomous driving. *Autonomous Vehicles and Machines, Electronic Imaging*, 2017.

[10] Shixiang Gu, Timothy Lillicrap, Ilya Sutskever, and Sergey Levine. Continuous deep q-learning with model-based acceleration. *arXiv preprint arXiv:1603.00748*, 2016.

[11] Xiaoxiao Guo, Satinder Singh, Honglak Lee, Richard L Lewis, and Xiaoshi Wang. Deep learning for real-time atari game play using offline monte-carlo tree search planning. In *Advances in neural information processing systems*, pages 3338–3346, 2014.

[12] Jens Kober, J. Andrew Bagnell, and Jan Peters. Reinforcement learning in robotics: A survey. *I. J. Robotics Res.*, 32(11):1238–1274, 2013. doi: 10.1177/0278364913495721. URL http://dx.doi.org/10.1177/0278364913495721.

[13] Timothy P Lillicrap, Jonathan J Hunt, Alexander Pritzel, Nicolas Heess, Tom Erez, Yuval Tassa, David Silver, and Daan Wierstra. Continuous control with deep reinforcement learning. *arXiv preprint arXiv:1509.02971*, 2015.

[14] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. Playing atari with deep reinforcement learning. *arXiv preprint arXiv:1312.5602*, 2013.

[15] Volodymyr Mnih, Adria Puigdomenech Badia, Mehdi Mirza, Alex Graves, Timothy P Lillicrap, Tim Harley, David Silver, and Koray Kavukcuoglu. Asynchronous methods for deep reinforcement learning. In *International Conference on Machine Learning*, 2016.

[16] Erion Plaku and Sertac Karaman. Motion planning with temporal-logic specifications: Progress and challenges. *AI Communications*, (Preprint):1–12.

[17] Matthias Plappert. keras-rl. https://github.com/matthiasplappert/keras-rl, 2016.

[18] Vasumathi Raman, Constantine Lignos, Cameron Finucane, Kenton C. T. Lee, Mitch Marcus, and Hadas Kress-Gazit. Sorry dave, i'm afraid i can't do that: Explaining unachievable robot tasks using natural language. In *Robotics: Science and Systems (RSS)*, Berlin, Germany, 2013.

[19] Shai Shalev-Shwartz, Shaked Shammah, and Amnon Shashua. Safe, multi-agent, reinforcement learning for autonomous driving. *arXiv preprint arXiv:1610.03295*, 2016.

[20] Vikas Shivashankar, Krishnanand N Kaipa, Dana S Nau, and Satyandra K Gupta. Towards integrating hierarchical goal networks and motion planners to support planning for human-robot teams. 2014.

[21] David Silver, Aja Huang, Chris J Maddison, Arthur Guez, Laurent Sifre, George Van Den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, et al. Mastering the game of go with deep neural networks and tree search. *Nature*, 529 (7587):484–489, 2016.

[22] Marc Toussaint. Logic-geometric programming: An optimization-based approach to combined task and motion planning. In *International Joint Conference on Artificial Intelligence*, 2015.

[23] Huazhe Xu, Yang Gao, Fisher Yu, and Trevor Darrell. End-to-end learning of driving models from large-scale video datasets. *arXiv preprint arXiv:1612.01079*, 2016.

[24] Timothy Yee, Viliam Lisy, and Michael Bowling. Monte carlo tree search in continuous action spaces with execution uncertainty. IJCAI, 2016.