WiseMove: A Framework to Investigate Safe Deep Reinforcement Learning for Autonomous Driving

Jaeyoung Lee, Aravind Balakrishnan, Ashish Gaurav, Krzysztof Czarnecki(✉), and Sean Sedwards

University of Waterloo, Waterloo, Canada
kczarnec@gsd.uwaterloo.ca

Abstract. WiseMove is a platform to investigate safe deep reinforcement learning (DRL) in the context of motion planning for autonomous driving. It adopts a modular architecture that mirrors our autonomous vehicle software stack and can interleave learned and programmed components. Our initial investigation focuses on a state-of-the-art DRL approach from the literature, to quantify its safety and scalability in simulation, and thus evaluate its potential use on our vehicle.

1 Introduction

Ensuring the safety of learned components is of interest in many contexts and particularly in autonomous driving, which is the concern of our group.[1] We have hand-coded an autonomous driving motion planner that has already been used to drive autonomously for 100 km,[2] but we observe that further extensions by hand will be very labour-intensive. The success of deep reinforcement learning (DRL) in playing Go [5], and its success with other applications having intractable state space [1], suggests DRL as a more scalable way to implement motion planning. A recent DRL-based approach [4] seems particularly plausible, since it incorporates temporal logic (safety) constraints and its architecture is broadly similar to our existing software stack. The claimed results are promising, but the authors provide no means of verifying them and there is apparently no other platform in which to test their ideas. We have thus devised WiseMove, to quantify the trade-offs between safety, performance and scalability of both learned and programmed motion planning components.

Below we describe the key features of WiseMove and briefly present results of experiments that corroborate some of the claimed quantitative results of [4]. In contrast to that work, our results can be reproduced by installing our publicly-available code.[3]

[1] uwaterloo.ca/waterloo-intelligent-systems-engineering-lab/.
[2] therecord.com/news-story/8859691-waterloo-s-autonomoose-hits-100-kilometre-milestone/.
[3] git.uwaterloo.ca/wise-lab/wise-move/.

J. Lee, A. Balakrishnan, A. Gaurav and S. Sedwards—Contributed equally.

© Springer Nature Switzerland AG 2019
D. Parker and V. Wolf (Eds.): QEST 2019, LNCS 11785, pp. 350–354, 2019.
https://doi.org/10.1007/978-3-030-30281-8_20

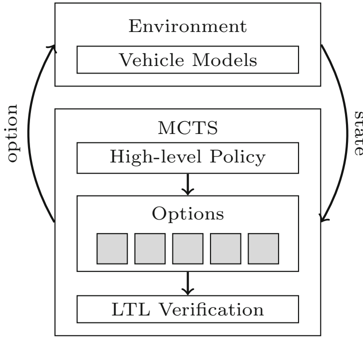

Fig. 1. Planning architecture.

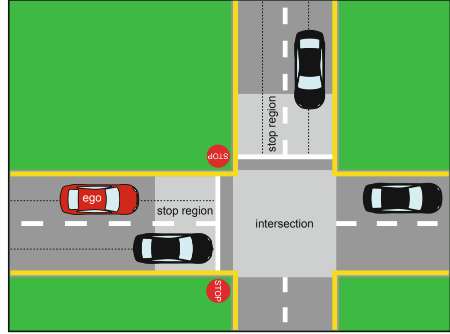

Fig. 2. Experimental environment.

2 Features and Architecture

WiseMove is an options-based modular DRL framework, written in Python, with a hierarchical structure designed to mirror the architecture of our autonomous driving software stack. Options [6, Chap. 17] are intended to model primitive manoeuvres, to which are associated low-level policies that implement them. A learned high-level policy over options decides which option to take in any given situation, while Monte Carlo tree search (MCTS [6, Chap. 8]) is used to improve overall performance during deployment (planning). High-level policies correspond to the *behaviour planner* in our software stack, while low-level policies correspond to the *local planner*. These standard concepts are discussed in, e.g., [3]. To define correct behaviour and option termination conditions, WiseMove incorporates "learntime" verification to validate individual simulation traces and assign rewards during both learning and planning. This typically improves safety, but does not guarantee it, given finite training and function approximation [1,2].

When an option is chosen by the decision maker (the high-level policy or MCTS), a sequence of actions is generated according to the option's low-level policy. An option terminates if there is a violation of a logical requirement, a collision, a timeout, or successful completion. In the latter case, the decision maker then chooses the next option to execute, and so on until the whole episode ends. Fig. 1 gives a diagrammatic overview of WiseMove's planning architecture. The current state is provided by the environment. The planning algorithm (MCTS) explores and verifies hypothesized future trajectories using the learned high-level policy as a baseline. MCTS chooses the best next option it discovers, which is then used to update the environment.

WiseMove comprises four high-level Python modules: `worlds`, `options`, `backends` and `verifier`. The `worlds` module provides support for environments that adhere to the OpenAI Gym[4] interface, which includes methods to initialize, update and visualize the environment, among others. The `options` module

[4] gym.openai.com.

Table 1. Options and examples of their preconditions.

Option	Description	Example LTL precondition
`KeepLane`	Keep lane while driving	-
`Stop`	Stop in the stop region	`G(not has_stopped_in_stop_region)`
`Wait`	Wait in the stop region then drive forward	`G((has_stopped_in_stop_region and in_stop_region) U highest_priority)`
`Follow`	Follow vehicle ahead	`G(veh_ahead U (in_stop_region or close_to_stop_region))`
`ChangeLane`	Change to other lane	`G(not(in_intersection or in_stop_region))`

defines the hierarchical decision-making structure. The `backends` module provides the code that implements the learned or possibly programmed components of the hierarchy. WiseMove currently uses `keras`[5] and `keras-rl`[6] for DRL training. The training hierarchy can be specified through a `json` file.

The `verifier` module provides methods for checking LTL-like properties constructed according to the following syntax:

$$\varphi = \mathtt{F}\ \varphi \mid \mathtt{G}\ \varphi \mid \mathtt{X}\ \varphi \mid \varphi\ \mathtt{=>}\ \varphi \mid \varphi\ \mathtt{or}\ \varphi \mid \varphi\ \mathtt{and}\ \varphi \mid \mathtt{not}\ \varphi \mid \varphi\ \mathtt{U}\ \varphi \mid \mathtt{(}\varphi\mathtt{)} \mid \alpha \tag{1}$$

Atomic propositions, α, are functions of the global state, represented by human-readable strings. In what follows we use the term LTL to mean properties written according to (1). The verifier decides during learning and planning when various LTL properties are satisfied or violated, in order to assign the appropriate reward. Learning proceeds one step at a time, so the verifier works incrementally, without revisiting the prefix of a trace. WiseMove uses LTL to express the preconditions and terminal conditions of each option, as well as to encode traffic rules. E.g.,

`G(in_stop_region => (in_stop_region U has_stopped_in_stop_region)).`

Some options and preconditions are listed in Table 1.

3 Experiments

Our experiments reproduce the architecture and some of the results of [4],[7] using the scenario illustrated in Fig. 2. We learned the low-level policies for each option first, then learned the high-level policy that determines which option to use at each decision instant. We used the DDPG [1] and DQN [2] algorithms to learn the low- and high-level polices, respectively. Each episode is initialized with the

[5] keras.io.

[6] github.com/keras-rl/keras-rl.

[7] Details and scripts to reproduce our results can be found in our repository (see Footnote 3).

ego vehicle placed at the left hand side, and up to six other randomly placed vehicles driving “aggressively” [4]. The goal of the ego is to reach the right hand side with no collisions and no LTL violations.

Table 2. Performance of low-level policies trained for 10^5 steps, with and without additional LTL: mean (std) % success, averaged over 100 trials of 100 episodes.

Add'l LTL	`KeepLane`	`Stop`	`Wait`	`Follow`	`ChangeLane`
Without	7.7 (21.4)	53.9 (32.0)	36.7 (43.3)	52.0 (19.6)	60.9 (34.0)
With	78.1 (29.4)	87.6 (20.4)	78.3 (28.8)	81.0 (15.4)	92.8 (14.3)

Table 3. Overall performance with and without MCTS, using low-level policies trained for 10^6 steps: mean (std) %, averaged over 1000 episodes.

Without MCTS			With MCTS		
Success	LTL violation	Collision	Success	LTL violation	Collision
92.0 (2.0)	5.40 (1.9)	2.60 (1.6)	98.5 (1.5)	0.9 (0.9)	0.6 (0.8)

We found that training low-level policies only according to the information given in [4] is unreliable; training would often not converge and good policies had to be selected from multiple attempts. We thus introduced additional LTL to give more information to the agent during training, including liveness constraints (e.g., `G(not stopped_now)`) to promote exploration, and safety-related properties (e.g., `G(not veh_ahead_too_close)`). Table 2 reports typical performance gains for 10^5 training steps. Note in particular the sharp increase in performance for `KeepLane`, which is principally due to the addition of a liveness constraint. Without this, the agent avoids the high penalty of collisions by simply waiting, thus not completing the option.

Having trained good low-level policies with DDPG using 10^6 steps, we trained high-level policies with DQN using 2×10^5 steps. We then tested the policies with and without MCTS. Table 3 reports the results, which suggest a ca. 7% improvement using MCTS.

4 Conclusion and Prospects

We have constructed WiseMove to investigate safe deep reinforcement learning in the context of autonomous driving. Learning is via options, whose low- and high-level policies broadly mirror the behaviour planner and local planner in our autonomous driving stack. The learned policies are deployed using a Monte Carlo tree search planning algorithm, which adapts the policies to situations that may not have been encountered during training. During both learning and planning, WiseMove uses linear temporal logic to enable and terminate options, and to specify safe and desirable behaviour.

Our initial investigation using WiseMove has reproduced some of the quantitative results of [4]. To achieve these we found it necessary to use additional logical constraints that are not mentioned in [4]. These enhance training by promoting exploration and generally encouraging good behaviour. We leave a detailed analysis for future work.

Our ongoing research will use WiseMove with different scenarios and more complex vehicle dynamics. We will also use different types of non-ego vehicles (aggressive, passive, learned, programmed, etc.) and interleave learned components with programmed components from our autonomous driving stack.

Acknowledgment. This work is supported by the Japanese Science and Technology agency (JST) ERATO project JPMJER1603: HASUO Metamathematics for Systems Design, and by the Natural Sciences and Engineering Research Council of Canada (NSERC) Discovery Grant: Model-Based Synthesis and Safety Assurance of Intelligent Controllers for Autonomous Vehicles.

References

1. Lillicrap, T.P., et al.: Continuous control with deep reinforcement learning (2015). http://arxiv.org/abs/1509.02971
2. Mnih, V., et al.: Playing Atari with deep reinforcement learning (2013). http://arxiv.org/abs/11312.5602
3. Paden, B., Čáp, M., Yong, S.Z., Yershov, D., Frazzoli, E.: A survey of motion planning and control techniques for self-driving urban vehicles. IEEE Trans. Intell. Veh. **1**(1), 33–55 (2016)
4. Paxton, C., Raman, V., Hager, G.D., Kobilarov, M.: Combining neural networks and tree search for task and motion planning in challenging environments. In: 2017 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS), IEEE, pp. 6059–6066 (2017)
5. Silver, D., et al.: Mastering the game of Go with deep neural networks and tree search. Nature **529**, 484–489 (2016)
6. Sutton, R.S., Barto, A.G.: Reinforcement Learning: An Introduction. MIT Press, Cambridge (2018)