A Framework for DRL Navigation With State Transition Checking and Velocity Increment Scheduling

CHANDRA KUSUMA DEWA\textsuperscript{1,2} AND JUN MIURA\textsuperscript{1}, (Member, IEEE)

\textsuperscript{1}Department of Computer Science and Engineering, Toyohashi University of Technology, Toyohashi 441-8580, Japan
\textsuperscript{2}Department of Informatics, Faculty of Industrial Technology, Universitas Islam Indonesia, Yogyakarta 55584, Indonesia

Corresponding author: Chandra Kusuma Dewa (chandra.kusuma.dewa.mm@tut.jp; chandra.kusuma@uii.ac.id)

This work was supported in part by the Japan Society for the Promotion of Science (JSPS) KAKENHI under Grant 17H01799.

ABSTRACT

To train a mobile robot to navigate using end-to-end approach which maps sensors data into actions, we can use deep reinforcement learning (DRL) method by providing training environments with proper reward functions. Although some studies have shown the success of DRL in navigation task for mobile robots, the method needs appropriate hyperparameter settings such as the environment’s timestep size and the robot’s velocity range to produce a good navigation policy. The previous existing DRL framework has proposed the use of odometry sensor to generate dynamic timestep size in the environment to solve the mismatch problem between the timestep size and the robot’s velocity. However, the framework lacks a procedure for checking terminal conditions which may occur during action executions resulting inconsistency in the environment and poor navigation policies. In the case of navigation task, terminal conditions may happen when the robot achieves the navigation goal position or collides with obstacles while performing an action in one timestep. To cope with this problem, we propose a state transition checking method in the DRL environment which is specific for navigation task that leverages odometry and laser sensor to ensure that the environment follows Markov Decision Process with dynamic timestep size. We also introduce a velocity increment scheduling to stabilize the mobile robot during training. Our experiment results show that state transition checking along with the velocity increment scheduling are able to make the robot navigate faster with higher success rate compared to other existing DRL frameworks.

INDEX TERMS

Deep reinforcement learning, state transition checking, velocity increment scheduling, robot navigation.

I. INTRODUCTION

Recently, reinforcement learning (RL) has been gaining some interest and has been widely implemented in various fields, including robotics [1]. We can train policies for a robot to generate desired behaviors for specific tasks by creating training environments for the RL agent which provide appropriate reward functions using RL framework. The RL agent will then learn to produce appropriate actions for the robot based on current environment’s states by maximizing expected future rewards through interactions with the environments [2]. With the advancements and potentials of deep learning [3], the concept of deep reinforcement learning (DRL) then exists which extends traditional RL framework with deep learning methods. DRL is able to make the agent to generate more appropriate actions since it enables better feature extractions of the environment by leveraging deeper neural network architectures with wider scope and variety of state representations [4], [5].

In navigation tasks, DRL is proven to be successfully implemented to train agents to perform path planning [6], [7]. It also offers some advantages in training a mobile robot to avoid obstacles and also to achieve navigation destination points. Provided training environments which follow Markov Decision Process (MDP), we only need suitable state representation and also appropriate rewards as the input for the DRL agent to generate the velocity of the robot to navigate in the training process [8]. We also do not have to collect
enormous labeled data to train navigation policies, as in the training procedure in supervised learning since the robot will learn from interactions with the training environments [9]. As mentioned in [10], [11], and [12], it is also possible to use the learned policies for the mobile robot to navigate in new unknown complex environments without having to retrain the DRL agent.

However, challenges in DRL method do exist [13], [14]. In order to get an optimal policy to be applied for mobile robots to navigate using the method, we need to appropriately implement the training environment and also carefully set the proper hyperparameters [15]. One of the hyperparameters that should be correctly set in the DRL environment is the timestep size which indicates the duration of a single action (can be selected and performed) in the environment [16]. Wrongly chosen timestep size will cause poor performance since the Q-function may collapse to V-function so that the DRL agent will not get enough information to select actions resulting higher reward [17]. In the case of navigation task training with DRL, the second problem which may occur is how we can train the DRL agent to generate velocities which can make a robot to navigate quickly but still considered as safe since faster navigation may cause higher collision rate.

A new way to solve the timestep size problem in DRL environment has been introduced by openai_ros (http://wiki.ros.org/openai_ros) which sets dynamic timestep size for each action to be performed in the DRL environment. The openai_ros package sets the duration of each action by comparing the output velocity from the DRL agent to the current robot’s velocity from odometry sensor so that the timestep size will always match with the robot’s velocity. Nevertheless, the package always assumes that all actions will always be successfully executed in the environment and does not put much attention on terminal conditions which may occur in the middle of performing an action. In the case of robot navigation, terminal conditions may happen when the robot achieves the navigation goal position or collides with obstacles during an action execution. When the terminal conditions occur, the environment should move to terminal state and a training episode should be ended. Otherwise, the condition may lead to inconsistency in the environment which can make the training process fail.

To solve the second problem, several studies such as [18] and [19] include the DRL agent’s output action, which is the velocity, as the magnitude for the reward function to make it able to generate high velocity value for autonomous outdoor vehicles. In the context of DRL based robot navigation task, although the agent is forced to generate higher velocity value as in [20] and [21], only small values are set for the robot’s maximum velocities which prevent it from navigating quickly. Even though [18] demonstrates that the DRL agent can generate smaller velocity towards obstacles, the aforementioned studies mostly focus on how to make the agent generate faster velocity rather than consider how the robot can navigate safely in the environment using the learned policy.

The goal of our study is to train a mobile robot to navigate fast yet safe in indoor environments using DRL method. In our study, we focus on DRL navigation for a mobile robot which is equipped with laser sensor to detect obstacles and odometry sensor to estimate the robot’s velocity. In order to accomplish our objective, we propose a novel framework for DRL based navigation which extends the ordinary RL mechanism and previously existing frameworks by applying state transition checking in the DRL environment to ensure that it will always follow MDP. Moreover, we also apply velocity increment scheduling which is based on curriculum learning for the mobile robot during the training process of the DRL agent as depicted in Fig. 1. The main contributions of the paper are summarized as follows:

1) In our proposed state transition checking method, we extend openai_ros package so that it becomes aware of terminal conditions which may occur in the middle of performing an action in the environment. The method continuously checks whether the current state should move to terminal state whenever the DRL agent meets terminal conditions which may happen in the navigation task, such as arriving at the navigation goal position, colliding with obstacles, or reaching the maximum steps. The current state will then be immediately changed to terminal state when one of those conditions occurs. By doing so, we can ensure that the DRL environment will always follow MDP. We also do not have to manually set the timestep size in the environment since the duration of each action to be performed will be dynamically set not only by using odometry but also by using laser sensor.

2) In contrast to other methods, our proposed framework schedules the range of the robot’s velocity using a velocity increment scheduling mechanism to make a DRL agent learn to navigate fast. The scheduler will set the range of the robot’s velocity with a small value at the beginning of the training. It then will expand the range gradually along with the increasing number of the training episode. By using the mechanism, the DRL
C. K. Dewa, J. Miura: Framework for DRL Navigation With State Transition Checking and Velocity Increment Scheduling

**FIGURE 2.** The structure of Soft-Actor Critic (SAC) algorithm proposed by [31]. SAC employs five neural network functions which composed of policy network, Q network, and value network. The update process of all networks are done by using data which are sampled from the replay buffer for each episode iteration.

agent is given the opportunity to gently learn the skill to navigate fast but safe from the easiest level to the hardest level.

The remainder of the paper is organized as follows. Section II introduces several studies related to the fundamentals of DRL which underlies our proposed state transition checking method. In addition, references related to curriculum learning in RL which underlie our proposed velocity increment scheduling are also introduced in the section. Section III introduces our proposed framework in details including the state transition checking implementation in the DRL environment and also the velocity increment scheduling mechanism for the mobile robot in the training process. The verification of our proposed framework by simulations is provided in section IV. In section V, conclusions are drawn with discussion of the future work.

**II. RELATED WORK**

**A. DEEP REINFORCEMENT LEARNING**

DRL is part of machine learning techniques which is the intersection field between deep learning and RL [22]. By leveraging deep learning in RL, it is possible to train an agent to have better understanding of the environment where it interacts. Mnih et al. [23] demonstrated that DRL can be used to train an agent to have a human-level control skill in playing Atari games. Moreover, Silver et al. [24] showed that the trained DRL agent even can beat human skills in playing Go. In addition, several studies such as [25]–[27], and [28] have demonstrated that DRL can also be successively applied in robotics.

In order to train a DRL agent, an environment which follows Markov Decision Process (MDP) is needed. As described in Bellman [29], the environment should be able to be defined in tuple $$(S, A, p, r, \gamma)$$, where $S$ is state space, $A$ is action space, $p : S \times A \times S \rightarrow [0, 1]$ is state transition probability function, $r : S \times A \rightarrow [r_{\text{min}}, r_{\text{max}}]$ is bounded reward function, and $\gamma \in [0, 1]$ is the discount rate of reward. In the training process, the DRL agent will interact with the environment which always provides immediate reward $r_t$ whenever the agent performs an action $a_t \in A$ based on current state $s_t \in S$. The policy $\pi : S \rightarrow A$ is then learned by the agent by maximizing the sum of expected future reward using a specified DRL algorithm.

In our study, we focus on the details of the implementation of the DRL environment. As small changes in the environment may influence the resulting navigation policy [30], we need a mechanism which ensures that the environment will always follow MDP. In the case of navigation task, our proposed state transition checking method verifies that the environment follows MDP by inspecting terminal state conditions which may occur while actions are performed in the environment. Moreover, our method examines whether each action has been successfully executed in the environment by setting the timestep size in the environment dynamically.

**B. SOFT-ACTOR CRITIC**

Soft-Actor Critic (SAC) is one of model-free, off-policy DRL algorithms which was introduced by Haarnoja et al. [31]. Disimilar to other DRL algorithms which only consider maximizing expected future reward in the training process, SAC also aims to maximize entropy along with the future reward by utilizing a stochastic actor. As a result, the algorithm gives better performance since it can make the agent explores more in the environment compared to other algorithms which use deterministic actor architecture, such as Deep Deterministic Policy Gradient (DDPG) [32] and twin delayed deep deterministic (TD3) policy gradient algorithm [33].

As depicted in Fig. 2, we need three types of functions to implement SAC which are all implemented as neural
networks, namely policy networks, value networks, and Q networks. Since SAC is included as one of actor-critic frameworks in DRL algorithms, the policy network becomes the actor, whilst the value network along with the Q network become the critic. Additionally, SAC structure uses two Q networks to solve positive bias problem in the policy training process becomes more stable [23].

For each step iteration in time step \( t \), the policy network generates and performs an action \( a_t \) based on the current state of the environment \( s_t \) to obtain the next state \( s_{t+1} \) and the immediate reward \( r_t \). Subsequently, an experience which consists of tuple \((s_t, a_t, r_t, s_{t+1})\) is stored in the replay buffer \( D \). At the end of each training episode, the policy network and all of the critic networks are all updated by previous experiences sampled from \( D \). The updates are sequentially performed for 1) the value networks, 2) both Q networks, 3) the policy networks, and 4) the target value networks.

We choose to employ SAC in our study since it is one of the leading state-of-the-art methods which may improve the convergence ability and may able to avoid the high sample complexity problem in DRL training. We also choose to apply SAC since the algorithm can be appropriately used to train DRL agents performing task in environments with continuous action space. In the case of navigation task with differential drive robot, the output of the policy networks is the linear and the angular velocity \( a_t = (v_{net}^t, \omega_{net}^t) \) to be published to the mobile robot. However, SAC does not provide a mechanism to limit the range of \( v_{net} \) and \( \omega_{net} \) in the environment. Therefore, we need an additional clipping function to ensure that the final output actions from the RL agent always lay in a specified range.

C. CURRICULUM LEARNING

Bengio et al. [35] introduces curriculum learning as a training strategy in the context of machine learning which presents organized examples based on meaningful order, gradually from simple to complex. As reported in Hacohen and Weinshall [36] and also Narvekar [37], it is possible to make the training process of deep neural networks and RL becomes faster with improved final performance using this strategy. However, generating appropriate examples following a specified curriculum is considered to be an open problem to be solved [38].

In the context of DRL, curriculum learning strategy is used to generate suitable experiences to be sampled from the replay buffer in the update process of neural networks. Schaul et al. [39] introduces Prioritize Experience Replay (PER) which priorities important experiences to be sampled from the replay buffer to generate the examples following curriculum learning scheme. Subsequently, Ren et al. [40] combines self-paced prioritize function and coverage penalty function which could select samples with appropriate difficulty with penalty when samples are replayed frequently. Another studies, such as [41] and [42] use curriculum learning to schedule ordered list of task and maps to be solved by the RL agent.

We consider that generating actions for a mobile robot to navigate fast yet safe is a complex task. From the perspective of curriculum learning, we see that the task can be gradually learned from simple tasks. Furthermore, we relate the difficulty of the navigation task to the robot’s velocity range settings. We first set a small range in the robot velocity setting at the initial training stage. Subsequently, we follow the curriculum learning strategy by expanding the velocity range during the training process gradually.

III. DRL-BASED NAVIGATION FRAMEWORK WITH STATE TRANSITION CHECKING AND VELOCITY INCREMENT SCHEDULING

The details of our proposed framework are presented in this section. We first describe the properties of the DRL environment which we use to train the agent for the navigation task. Afterwards, we describe our proposed state transition checking in the environment and our proposed velocity increment scheduling for the mobile robot in the training process.

A. DRL-BASED NAVIGATION ENVIRONMENT

We follow the definition of MDP to describe our environment for training a mobile robot to navigate in an indoor setting as outlined in the following subsections.

1) STATE SPACE

We need to carefully design the state space in the DRL environment since it will greatly affect the generated actions by the agent. To make the DRL agent able to generate appropriate actions to navigate, it needs information related to the goal position and the surrounding obstacles from the laser sensor attached to the mobile robot. Additionally, it also needs information related to the current robot’s state which is represented as the previous velocity generated by the agent which has been successfully executed in the environment. Therefore, the state space in this study consists of laser scan data from the laser sensor, the goal coordinate position, and also the robot’s previous velocity [43].

Fig. 3 depicts variables that we use to formulate the state space. Suppose that the robot’s current state is at the linear and angular velocity of \( v_{t-1} \) and \( \omega_{t-1} \), we represent the navigation goal position \( p_{goal} \) as the relative coordinate of \( p_x \) and \( p_y \) from the robot’s current position. In contrast to [44], we do not directly use all of the laser scan values to represent the surrounding obstacles around the robot. We follow [45] and [46] who divide the laser scans into sectors and then only include values which represent all sectors in the state space. Therefore, the state space \( s \in S \) is defined as,

\[
S = \{ L_{scr,t} | p_x, p_y, v_{t-1}, \omega_{t-1} \},
\]

where \( L_{scr,t} = \{ l_{scr,i} | i \in 1 \ldots M \} \) denotes all of the sector’s laser scans and \( M \) denotes the number of sector.
To compute \( L_{\text{scstr}} \), as also depicted in Fig. 3, we first adjust \( N \) laser scans obtained from the laser sensor \( L = \{ l_j \mid j \in 1 \ldots N \} \) by limiting the values to a radius constraint of \( l_{\text{max}} \) so that the RL agent can more easily portray the state space in the environment. The function to compute the adjusted laser scan values \( l_{\text{adj}} \) is given by

\[
l_{\text{adj},k} = \begin{cases} l_j, & l_j \leq l_{\text{max}} \\ l_{\text{max}}, & l_j > l_{\text{max}}, \end{cases}
\]

where \( k \in \{1, 2, \ldots, N \} \). Subsequently, we compute the average values of the adjusted laser scans for all sectors using the following formula:

\[
l_{\text{scstr}} = \frac{\sum_{z=0}^{\lfloor M/2 \rfloor} l_{\text{adj},z}}{M/2}.
\]

We also define a terminal state which ends an episode in the training or validation process. Current state will move to terminal state \( s_T \) whenever the robot successfully arrives at a defined goal position, collides with barriers, or reaches the allowed maximum steps in each episode.

2) ACTION SPACE

We use continuous action space \([23]\) so that the agent will generate more dimension output actions resulting more smooth robot’s trajectories when it navigates \([47]\). The continuous action space \( a \in \mathcal{A} \) is defined as

\[
\mathcal{A} = \{ v, \omega \},
\]

where \( v \) and \( \omega \) denote linear and angular velocity to be published to the mobile robot respectively. Here, we assume that the robot in the environment is a differential drive robot which can be controlled by publishing \( v \) and \( \omega \).

To control the generated output action so that it will always follow ranges of \( v \in [v_{\text{min}}, v_{\text{max}}] \) and \( \omega \in [\omega_{\text{min}}, \omega_{\text{max}}] \) for time step \( t \), we adjust the velocities as follows

\[
v_{\text{adj}} = \frac{v_{\text{min}} + (v_{\text{net}} + 1)(v_{\text{max}} - v_{\text{min}})}{2},
\]

\[
\omega_{\text{adj}} = \frac{\omega_{\text{min}} + (\omega_{\text{net}} + 1)(\omega_{\text{max}} - \omega_{\text{min}})}{2},
\]

where \( v_{\text{adj}} \) and \( \omega_{\text{adj}} \) denote the adjusted velocities, whilst \( v_{\text{net}} \) and \( \omega_{\text{net}} \) denote the velocities generated from the RL agent (the policy networks). Finally, we clip the action using the following functions:

\[
v_t = \begin{cases} v_{\text{max}}, & v_{\text{adj}} > v_{\text{max}} \\ v_{\text{adj}}, & v_{\text{min}} \leq v_{\text{adj}} \leq v_{\text{max}} \\ v_{\text{min}}, & v_{\text{adj}} < v_{\text{min}} \end{cases}
\]

\[
\omega_t = \begin{cases} \omega_{\text{max}}, & \omega_{\text{adj}} > \omega_{\text{max}} \\ \omega_{\text{adj}}, & \omega_{\text{min}} \leq \omega_{\text{adj}} \leq \omega_{\text{max}} \\ \omega_{\text{min}}, & \omega_{\text{adj}} < \omega_{\text{min}} \end{cases}
\]

3) REWARD FUNCTION

We follow \([46]\) which designs the reward function for the navigation task based on the artificial potential field method \([48]\). Using the function, the RL agent will be given a positive reward \( c_{\text{goal}} \) when the robot is able to arrive at the specified destination point. A negative penalty reward \(-c_{\text{crash}}\) will also be given to the agent whenever the robot collides with obstacles. Additionally, the reward obtained by the agent will be computed based on the attractive potential field \( U_{\text{attr}} \) towards the goal position and the repulsive potential field \( U_{\text{rep}} \) towards surrounding obstacles.

The function to calculate the immediate reward for the time step \( t \) is defined as,

\[
r_t = \begin{cases} k_{\text{goal}}, & d_{\text{goal}} < \epsilon_{\text{goal}} \\ -k_{\text{crash}}, & d_{\text{crash}} < \epsilon_{\text{crash}} \\ -U_{\text{attr}} - U_{\text{rep}} - r_{t-1}, & \text{otherwise}, \end{cases}
\]

where \( d_{\text{goal}} \) and \( d_{\text{crash}} \) denote the current robot’s distance to the goal position and the nearest distance to an obstacle. Here, \( \epsilon_{\text{goal}} \) and \( \epsilon_{\text{crash}} \) denote small distance tolerance values toward the goal position and toward an obstacle. Note that the function also includes reward from the previous time step \( r_{t-1} \) to force the robot moving faster to reach the goal position.

We use the following formula to compute \( U_{\text{attr}} \) in the reward function,

\[
U_{\text{attr}} = k_{\text{attr}} \ast d_{\text{goal}},
\]

where \( k_{\text{attr}} \) denotes the constant for the attractive field. On the other hand, we use the following formula to compute the repulsive force:

\[
U_{\text{rep}} = k_{\text{rep}} \sum_{i=1}^{N} \left( \frac{1}{c + \frac{d_{\text{map,obs}}}{e}} \right) - \frac{k_{\text{rep}} \ast N}{c + l_{\text{max}}},
\]
where $k_{\text{rep}}$ denotes the constant for the repulsive field and $N$ denotes the number of surrounding obstacles. Here, $d_{\text{map-obs}}$ denotes the distance of the robot to the detected obstacle $i$ in the local map whilst $c$ and $l_{\text{max}}$ denote a constant to limit the repulsive field and the maximum laser scan range value respectively.

### B. STATE TRANSITION CHECKING

In this article, we propose a state transition checking procedure in an RL environment which is specifically intended for navigation task by leveraging odometry and laser sensor attached to the mobile robot. In our proposed method, the odometry sensor is utilized to automatically determine the dynamic timestep size which is needed to successfully perform each action in the environment. Furthermore, the laser sensor is employed to detect whether the current state should move to terminal state while an action is performed in the environment. In addition, the data from the laser sensor and the odometry sensor are directly used in our method instead of modeling them.

We notice that, specific for the navigation task, current state may move to terminal state before an action can be successfully performed in the environment. This condition may happen since the robot may arrive at the goal position or may collide with obstacles in the middle of performing an action and before the action is considered as done to be executed in the environment. When the condition occurs, the action should be cut and the environment’s state should be moved to terminal state to keep the consistency in the environment.

Fig. 4 depicts the illustration of our proposed state transition checking method in the RL environment. The proposed checking function is represented as yellow rounded rectangles which will continuously check whether an action has been successfully performed in the environment or whether one of the terminal conditions occurs by using data obtained from odometry and laser sensor attached to the mobile robot. Note that the timestep size in the environment will be dynamically assigned as the width of the yellow rounded rectangle may vary for each action as shown in the case of action $a_0$, $a_1$, and $a_2$. In the case of action $a_3$, the state transition checking function is able to cut the action and capable to prevent the environment to move from state $s_3$ to state $s_4$. Instead, the environment moves from state $s_3$ to terminal state $s_T$.

The detailed formulation of our proposed state transition checking method is given by

$$s_{t+1} \sim \begin{cases} s_T, & d_{\text{obs}} < \epsilon_{\text{obs}} \text{ or } d_{\text{goal}} < \epsilon_{\text{goal}} \text{ or } t = N \\ \kappa(s_t), & \text{otherwise} \end{cases}$$

Here, the next state $s_{t+1}$ becomes the terminal state $s_T$ whenever the robot collides with obstacles ($d_{\text{obs}} < \epsilon_{\text{obs}}$), arrives at the navigation goal position ($d_{\text{goal}} < \epsilon_{\text{goal}}$), or when the maximum step in one episode is reached ($t = N$). Suppose that we train the RL agent using SAC, the environment will move to the next state if the robot’s current velocity obtained from odometry sensor ($v_{\text{odom}}$, $\omega_{\text{odom}}$) is close to the output action from the RL agent ($v_t$, $\omega_t$), as described in the following formula:

$$\kappa(s_t) = p(s_{t+1} | s_t, \{v_t, \omega_t\}) \pi_{\phi}(\{v_t, \omega_t\} | s_t),$$

where $\pi_{\phi}$ denotes the stochastic policy which generates the probability of taking action $a_t = \{v_t, \omega_t\}$ in state $s_t$. Moreover, $p$ denotes the transition probability function in the RL environment, given the current state $s_t$ and current action $a_t$.

In more details, we define Algorithm 1 to implement our proposed state transition checking procedure in an RL environment. The proposed algorithm extends openai_ros package and the standardized step function in OpenAI Gym framework [49] which performs the generated output action from the RL agent to the environment. As shown in line 3 through line 27, the algorithm utilizes a main looping procedure which will publish the velocity of the robot (line 4), obtain data from odometry and laser sensor (line 5 and line 6), and check the terminal conditions (line 9) continuously. Whenever one of the terminal conditions occurs based on the data from laser sensor, the looping procedure will be stopped and the environment will move to terminal state (line 16). Otherwise, the looping will only be stopped if the current robot’s velocity obtained from the odometry sensor $\{v_{\text{odom}}, \omega_{\text{odom}}\}$ is equal to the output action generated by the RL agent $a_t = \{v_t, \omega_t\}$ with a velocity difference tolerance $\xi$ (line 19). In addition, the robot will be stopped whenever the allowed maximum steps in one episode is reached (line 20).

### C. VELOCITY INCREMENT SCHEDULING

During the training process, we propose velocity increment scheduling for the mobile robot along with our proposed state
transition checking method to make the robot can navigate faster while still maintaining high value of success rate. Our proposed scheduling technique is one implementations of curriculum learning which manages the range value of the robot’s velocity. In the early stage of the training process, a small velocity range is initially set. Subsequently, as the number of training episodes rises, we gradually increase the range of the robot’s velocity linearly. Therefore, the complex task of generating appropriate actions which enables the mobile robot to navigate fast yet safe can be gradually and gently learned by the RL agent.

Suppose that we want to schedule the robot’s velocity from lower speed A to higher speed B during N training episodes. Additionally, speed A has range of linear velocity of \( v_a \in [v_{a_{min}}, v_{a_{max}}] \) and range of angular velocity of \( \omega_a \in [\omega_{a_{min}}, \omega_{a_{max}}] \), whilst speed B has range of linear velocity of \( v_b \in [v_{b_{min}}, v_{b_{max}}] \) and range of angular velocity of \( \omega_b \in [\omega_{b_{min}}, \omega_{b_{max}}] \). Using our velocity increment scheduling technique, the range of the linear velocity for the current episode \( \epsilon \) is given by

\[
v_{\epsilon} = [v_{\epsilon_{min}} v_{\epsilon_{max}} + \epsilon * \Delta v_{\max}],
\]

where \( \Delta v_{\max} = (v_{\max} - v_{\min})/N \) denotes the increment of the higher bound of the linear velocity for each episode. Here, we do not expand the range of the lower bound of the linear velocity since it is always set to zero and we do not want the robot to move backward while navigating. Similarly, the range of the angular velocity for the current episode \( \epsilon \) is given by

\[
\omega_{\epsilon} = [\omega_{\epsilon_{min}} - \epsilon * \Delta \omega_{\min}, \omega_{\epsilon_{max}} + \epsilon * \Delta \omega_{\max}],
\]

where \( \Delta \omega_{\min} = (\omega_{\min} - \omega_{\max})/N \) denotes the increment of the lower bound of the angular velocity for each episode and \( \Delta \omega_{\max} = (\omega_{\max} - \omega_{\min})/N \) denotes the increment of the higher bound of the angular velocity for each episode.

### IV. EXPERIMENT RESULTS AND DISCUSSIONS

In this section, detailed analysis and discussion of the experiment results for our proposed framework is presented. We first describe the experiment setup including the implementation and the baselines for comparison. Subsequently, we present our analysis towards the experiment results.

#### A. EXPERIMENT SETUP

1) IMPLEMENTATION

In order to validate our proposed framework, we conduct several experiments to train DRL agents to generate actions for a robot to navigate in an indoor environment. We implement our framework inside the Gazebo simulator [50] with a differential drive based Fetch robot [51] along with the Robot Operating System (ROS) [52] to subscribe data from the laser sensor and the odometry sensor. By using the ROS application programming interface (API), we directly feed the data from the sensors to the state transition checking method in our proposed framework. All of the navigation training environments used in this study are coded in Python which follow the standardized OpenAI Gym framework [49] and run on a workstation with an Nvidia Titan RTX graphic processor.

Fig. 5 depicts the maps which we use to train and to validate all of the DRL agents in this study. In the training process, we utilize a very simple map (Fig. 5 (a)) containing only four static cylinder obstacles. Additionally, we employ a more complex map (Fig. 5 (b)) in the validation process containing new static obstacles namely several boxes and several static human models. For both maps, we set the same one initial robot position and a goal position which is randomly picked from three candidates for each episode.
FIGURE 5. Maps used in the experiments. (a) Training map. (b) Validation map. Red circles represent the initial position of the mobile robot whilst green stars denote the navigation goal positions. For each episode, we set a random goal position for both maps.

FIGURE 6. Network architectures for the SAC algorithm used in this study. (a) Policy network. (b) Q Network. (c) Value network. The policy network serves as the actor, whilst the Q network and the Value network serve as critics.

All of the training process for the DRL agents are performed with SAC algorithm which is trained using Adam optimizer [53] and implemented using Pytorch library [54]. Moreover, we use network architectures which are shown in Fig. 6. We use the same two hidden layers with the same number of hidden neurons for each layer for all network architectures. In addition, the hyperparameters used in our experiments are listed in Table 1. We refer to [31] and [46] to define the hyperparameters for the SAC algorithm and for the RL environment with some additional adjustment. Furthermore, we refer to [43] to define the resolution of laser scans to compute values which represent each sector around the robot using Eq. (3).

To evaluate the performance of our proposed framework, we use the following metrics:

1) Success rate: the ratio of the number of goal position achieved to the number of episodes.

2) Average time to goal: the average time needed by the robot to arrive at the goal position. Here, we record time to goal values for all episodes when the robot can reach the navigation goal positions. Additionally, we set the time to goal values as zero for all episodes when the robot collides. Finally, we exclude all of the zero values to calculate the average time to goal value.

3) Episode ends: the number of the episode when the training process ends.

We train all of the DRL agents for 10,000 episodes and test all of the models for 2,000 episodes using various speed settings. We also set maximum step of 1,000 for each episode both in the training and in the validation process.

2) BASELINES
For comparison purpose, we train several DRL agents using other previously existing frameworks with SAC algorithm and with the same map. We compare our proposed framework with openai_ros and gym-gazebo which are a framework and a toolkit intended for training robots inside Gazebo simulator with ROS framework. Moreover, we also train DRL agents using our framework without the velocity increment scheduling. The followings are the baselines which we use for the comparison:

1) openai_ros. In contrast to our proposed framework which uses both odometry and laser sensor, the openai_ros package only uses odometry sensor to determine the dynamic timestep size in the DRL environment.

2) gym-gazebo. Unlike our proposed framework and the openai_ros package, the gym-gazebo toolkit [55] uses fixed time step in the DRL environment. In our

| Table 1. The hyperparameter settings of the SAC algorithm and the RL environment for the navigation task training in this study. |
|---------------------------------------------------------------|
| Hyperparameter | Setting         |
|-----------------|----------------|
| SAC Algorithm Parameter |               |
| Learning rate of all networks ($\eta$) | 0.0003         |
| Discount rate of reward ($\gamma$) | 0.99           |
| Soft update coefficient ($\tau$) | 0.01           |
| Batch size | 128            |
| Replay buffer size | 100,000        |
| RL Environment Parameter |               |
| Number of laser scan sectors ($|M|$) | 10             |
| Reward for arriving at the goal position ($r_{\text{goal}}$) | 100            |
| Distance to goal threshold in meter ($r_{\text{goal}}$) | 0.6            |
| Penalty for collisions ($r_{\text{coll}}$) | 100            |
| Min. distance to obstacle threshold in meter ($r_{\text{obs}}$) | 0.4            |
| Const. for attractive field ($k_{\text{attr}}$) | 100            |
| Const. for repulsive field ($k_{\text{rep}}$) | 2.0            |
| Const. to limit the repulsive field ($c$) | 0.04           |
| Const. to limit the max. laser scan range in meter ($l_{\text{max}}$) | 2.0            |
| Velocity difference threshold ($\xi$) | 0.3            |

VOLUME 8, 2020

191833
experiments, we set the same timestep size of 0.1s in the environment for all robot’s velocity setting scenarios.

3) Ours without VIS. We only use the state transition checking method in the environment without the velocity increment scheduling for the robot in the training process.

B. RESULTS AND ANALYSIS

The success rate values of training process in various speed settings are presented in Fig. 7. As depicted in Fig. 7 (a), our proposed framework which combines the state transition checking and the velocity increment scheduling gives high value of success rate. In addition, the training which schedules speed A to speed D gives better performance than the training which schedules speed A to C. Similarly, as shown in Fig. 7 (b), the success rate of our framework without the velocity increment scheduling with speed D is close to the success rate of the framework with speed A which has the lowest linear and angular velocity range value setting. Here, we confirm that lower range value of angular velocity can make the robot to navigate more stable.

Furthermore, we can see that the higher we set the velocity range value of the robot, the lower the success rate that we can get. As shown in Fig. 7 (b), we get the highest success rate when we set the robot at the lowest speed (speed A). When we increase the robot’s velocity setting to speed B or speed C, the success rates then decrease gradually. On the other hand, Fig. 7 (c) shows that the training process with the openai_ros package fails and stops at the very early stage when we set the robot with higher velocity range values since the robot never reaches the desired velocity obtained from odometry sensor when it collides with obstacles. As also shown in Fig. 7 (d), the mismatch problem of the timestep size and the robot’s velocity occurs with the gym-gazebo toolkit which uses fixed timestep size, resulting very unstable training process.

The progress of the time to goal value of the training process in various speed settings is presented in Fig. 8. By comparing Fig. 8 (a) to other sub figures, we can see that our proposed framework with the velocity increment scheduling is able to make the robot learn to move faster along with the increasing number of episodes. Moreover, the scheduling mechanism is proven able to reduce the average time to goal value. We get lower average time to goal values at the end of the training process for speed A in Fig. 8 (a) compared to the average time to goal value for speed A in Fig. 8 (b) and in Fig. 8 (c). Additionally, we also can see from Fig. 8 (a) that our proposed framework which schedules speed A to speed D is able to make the robot to have more stable average time to goal value compared to the training process which schedules speed A to speed C.

Table 2 shows the details of all experiments conducted in this study. We can see that our proposed framework which combines the state transition checking along with the velocity increment scheduling gives the best success rate both in the
FIGURE 8. The average time to goal of training process in various speed settings. (a) Our proposed framework. (b) Our proposed framework without the velocity increment scheduling. (c) The openai_ros package. (d) The gym-gazebo toolkit.

TABLE 2. Performance of our proposed framework in various speed settings. In both training and validation process, we confirm that the success rate of our proposed framework outperforms success rates of the baselines. Furthermore, we show that our framework can shorten the average time to goal while still maintaining a high success rate.

| Framework          | Speed | Training Success Rate (%) | Avg. Time to Goal (s) | Episode Ends | Validation Success Rate (%) | Avg. Time to Goal (s) |
|--------------------|-------|---------------------------|-----------------------|--------------|-----------------------------|-----------------------|
| Ours               | A→C   | 90.08                     | 26.48                 | 10,000       | 44.39                       | 19.08                 |
|                    | A→D   | 96.79                     | 23.19                 | 10,000       | 91.10                       | 16.63                 |
| Ours without VIS   | A     | 96.49                     | 32.92                 | 10,000       | 74.50                       | 27.47                 |
|                    | B     | 88.67                     | 29.56                 | 10,000       | 62.80                       | 23.79                 |
|                    | C     | 81.38                     | 20.53                 | 10,000       | 38.35                       | 19.18                 |
|                    | D     | 94.85                     | 16.77                 | 10,000       | 58.35                       | 14.20                 |
| openai_ros Package | A     | 96.55                     | 28.10                 | 10,000       | 73.09                       | 21.16                 |
|                    | B     | 36.95                     | 30.38                 | n/a          | n/a                         | n/a                   |
|                    | C     | 6.0                       | 12                    | n/a          | n/a                         | n/a                   |
|                    | D     | 6.66                      | 30.03                 | 15           | n/a                         | n/a                   |
| gym-gazebo Toolkit | A     | 2.14                      | 68.83                 | 10,000       | 0.0                         | n/a                   |
|                    | B     | 61.83                     | 23.8                  | 10,000       | 67.85                       | 21.89                 |
|                    | C     | 62.41                     | 15.16                 | 10,000       | 2.20                        | 15.99                 |
|                    | D     | 89.79                     | 12.83                 | 10,000       | 32.25                       | 12.89                 |

The stability and robustness of our proposed framework is also shown in Table 2. Our proposed state transition checking training and in the validation process compared to other frameworks in all robot’s velocity setting scenarios. On the other hand, the gym-gazebo toolkit gives the lowest average time to goal value among other frameworks. However, it results in lower success rate compared to our proposed framework both in the training and in the validation process. Therefore, we confirm that our framework can make the robot to navigate faster yet safe in the environment since it gives lower average time to goal values compared to other frameworks with speed A and speed B.

VOLUME 8, 2020 191835
in the RL environment is proven able to stabilize the training process with various robot’s velocity settings. The second block of the table lists the training results of our proposed framework without the velocity increment scheduling. From the list in the second block, we can clearly see that the success rate decreases and the average time to goal increases if we rise the robot’s velocity (speed A through speed C). On the other hand, experiment results with the gym-gazebo toolkit do not show the same pattern due to the mismatch problem between the timestep size in the environment and the robot’s velocity setting. Similarly, experiment results with the openai_ros package show that it does not robust to various robot’s velocity settings. In our experiments, the robot gets stuck on some obstacles and cannot finish the training for speed B, speed C, and speed D. From the results of our proposed framework, we can confirm that the framework can make the training process more stable and also able to be successfully conducted in various speed settings compared to other existing frameworks.

We also present plots of some generated navigation trajectories of the robot which is trained with our proposed framework in the validation map. As depicted in Fig. 9, we confirm that the higher we set the angular velocity range value of the robot, the more shaky the robot is. We can obviously see that we get very oscillate trajectories (denoted in blue lines) when we set the robot velocity with speed C in the training process. Including the velocity increment scheduling strategy in the training process gives better trajectories which can minimizes the collision risk (denoted in orange lines). Nevertheless, we still get oscillating trajectories as shown in the orange lines from Fig. 9 (b) and Fig. 9 (c).

On the contrary, we get more smooth trajectories if we set the robot with lower angular velocity in the training process. As we can see from Fig. 9, the trajectories generated from our proposed framework give better results when we schedule the velocity of the robot from speed A to speed D. Here, we only schedule the linear velocity in the training process. We can obviously see that the trajectories denoted as red lines are very similar to the trajectories denoted as green lines for all target settings in the validation map. This confirms that our proposed velocity increment scheduling strategy is able to make the robot to generate smooth trajectory as when we set the robot with speed A, but with higher linear velocity setting which can give better average time to goal value for the robot to navigate.

V. CONCLUSION

The paper proposed a framework for DRL based navigation for mobile robots with state transition checking in the RL environment and velocity increment scheduling for the robots during the training process. Based on the results in this study, we can conclude that the proposed framework performs well both in the training and in the validation process which enables the RL agent to learn a policy which can make the robot able to navigate fast yet safe in the environment. Moreover, we show that our proposed state transition checking in the RL environment is crucial to make the training process robust to various robot’s velocity settings. In addition, we show that our proposed velocity increment scheduling strategy is able to assist the RL agent to gently and gradually learn the complex task of navigating in an environment in high speed and with high success rate. The proposed framework has been validated in a simulated environment with various velocity settings which confirms that our proposed framework has better performance compared to other frameworks. For future work, we will include action smoothing strategy [19] in our framework to make the robot can generate more smooth trajectories for higher robot’s angular velocity settings in the training process.

REFERENCES

[1] J. Koher, J. A. Bagnell, and J. Peters, “Reinforcement learning in robotics: A survey,” Int. J. Robot. Res., vol. 32, no. 11, pp. 1238–1274, Sep. 2013.
[2] R. S. Sutton and A. G. Barto, Reinforcement Learning: An Introduction. Cambridge, MA, USA: Bradford Book, 2018.
[3] N. Sunderhauf, O. Brock, W. Scheirer, R. Hasdell, D. Fox, J. Leitner, B. Upcroft, P. Abbeel, W. Burgard, M. Milford, and P. Corke, “The limits and potentials of deep learning for robotics,” Int. J. Robot. Res., vol. 37, nos. 4–5, pp. 405–420, Apr. 2018.

[4] K. Arulkumaran, M. P. Deisenroth, M. Brundage, and A. A. Bharath, “Deep reinforcement learning: A brief survey,” IEEE Signal Process. Mag., vol. 34, no. 6, pp. 26–38, Nov. 2017.

[5] S. S. Moussavi, M. Schukat, and E. Howley, “Deep reinforcement learning: An overview,” in Proc. SAI Int. Conf. Syst. Chan, Switzerland: Springer, 2016, pp. 426–440.

[6] L. Lv, S. Zhang, D. Ding, and Y. Wang, “Path planning via an improved DQN-based learning policy,” IEEE Access, vol. 7, pp. 67319–67330, 2019.

[7] X. Zhou, P. Wu, H. Zhang, W. Guo, and Y. Liu, “Learn to navigate: Cooperative path planning for unmanned surface vehicles using deep reinforcement learning,” IEEE Access, vol. 7, pp. 165262–165278, 2019.

[8] J. Choi, K. Park, M. Kim, and S. Seok, “Deep reinforcement learning of navigation in a complex and crowded environment with a limited field of view,” in Proc. Int. Conf. Robot. Autom. (ICRA), May 2019, pp. 5993–6000.

[9] A. G. Barto and T. G. Dietterich, “Reinforcement learning and its relationship to supervised learning,” in Handbook of Learning and Approximate Dynamic Programming. Toronto, ON, Canada: Wiley, 2004, pp. 47–63.

[10] C. Wang, J. Wang, X. Zhang, and X. Zhang, “Autonomous navigation of UAV in large-scale unknown complex environment with deep reinforcement learning,” Proc. IEEE Global Conf. Signal Inf. Process. (GlobalSIP), Nov. 2017, pp. 858–862.

[11] A. R. Dooraki and D.-J. Lee, “An end-to-end deep reinforcement learning-based intelligent agent capable of autonomous exploration in unknown environments,” Sensors, vol. 18, no. 10, p. 3575, Oct. 2018.

[12] J. Zeng, R. Ju, L. Qin, Y. Hu, Q. Yin, and C. Hu, “Navigation in unknown dynamic environments based on deep reinforcement learning,” Sensors, vol. 19, no. 18, p. 3837, Sep. 2019.

[13] P. Kormushev, S. Calinon, and D. Caldwell, “Reinforcement learning in robotics: Applications and real-world challenges,” Robotics, vol. 2, no. 3, pp. 122–148, Jul. 2013.

[14] Z. Ding and H. Dong, “Challenges of reinforcement learning,” in Deep Reinforcement Learning. H. Dong, Z. Ding, and S. Zhang, Eds. Singapore: Springer, 2020, pp. 249–272.

[15] P. Henderson, R. Islam, P. Bachman, J. Pineau, D. Precup, and D. Meger, “Deep reinforcement learning that matters,” in Proc. 32nd AAAI Conf. Artif. Intell., 2018, pp. 3207–3214.

[16] S. Ramstedt and C. Pal, “Real-time reinforcement learning,” in Advances in Neural Information Processing Systems 32, H. Wallach, H. Larochelle, A. Beygelzimer, F. d’Alché-Buc, E. Fox, and R. Garnett, Eds. Red Hook, NY, USA: Curran Associates, 2019, pp. 3073–3082.

[17] C. Tallec, L. Blier, and Y. Ollivier, “Making deep Q-learning methods robust to time discretization,” in Proc. 36th Int. Conf. Mach. Learn., vol. 97, K. Chaudhuri and R. Salakhutdinov, Eds. Long Beach, CA, USA: PMLR, 2019, pp. 6096–6104.

[18] N. Deshpande and A. Spalanzani, “Deep reinforcement learning based vehicle navigation amongst pedestrians using a grid-based state representation,” in Proc. IEEE Intell. Transp. Syst. Conf. (ITSC), Oct. 2019, pp. 2081–2086.

[19] P. Cai, X. Mei, L. Tai, Y. Sun, and M. Liu, “High-speed autonomous drifting with deep reinforcement learning,” IEEE Robot. Autom. Lett., vol. 5, no. 2, pp. 1247–1254, Apr. 2020.

[20] L. Xie, S. Wang, A. Markham, and N. Trigoni, “Towards monocular vision field of view,” in Proc. Int. Conf. Robot. Autom. (ICRA), May 2019, pp. 484–503, 2016.

[21] D. Silver, A. Huang, J. C. Maddison, S. Dieleman, A. Guez, L. Sifre, G. V. D. Driessche, J. Schrittwieser, I. Antonoglou, M. Lanctot, D. Gwern, J. Nham, N. Kalchbrenner, I. Sutskever, T. Lillicrap, V. Panneershelvam, M. Leach, K. Kavukcuoglu, T. Graepel, and D. Hassabis, “Mastering the game of Go with deep neural networks and tree search,” Nature, vol. 529, pp. 484–503, Feb. 2015.
C. K. Dewa, J. Miura: Framework for DRL Navigation With State Transition Checking and Velocity Increment Scheduling

[45] M. Pfeiffer, S. Shukla, M. Turchetta, C. Cadena, A. Krause, R. Siegwart, and J. Nieto, “Reinforced imitation: Sample efficient deep reinforcement learning for mapless navigation by leveraging prior demonstrations,” IEEE Robot. Autom. Lett., vol. 3, no. 4, pp. 4423–4430, Oct. 2018.

[46] C. Sampredo, H. Bavle, A. Rodriguez-Ramos, P. D. L. Puente, and P. Campoy, “Laser-based reactive navigation for multirotor aerial robots using deep reinforcement learning,” in Proc. IEEE/RSJ Int. Conf. Intell. Robots Syst. (IROS), Madrid, Spain, Oct. 2018, pp. 1024–1031.

[47] L. Pang, Y. Zhang, S. Coleman, and H. Cao, “Efficient hybrid-supervised deep reinforcement learning for person following robot,” J. Intell. Robot. Syst., vol. 97, no. 2, pp. 299–312, Feb. 2020.

[48] O. Khatib, “Real-time obstacle avoidance for manipulators and mobile robots,” Int. J. Robot. Res., vol. 5, no. 1, pp. 90–98, Mar. 1986.

[49] G. Brockman, V. Cheung, L. Pettersson, J. Schneider, J. Schulman, J. Tang, and W. Zaremba. “OpenAI Gym,” 2016, arXiv:1606.01540. [Online]. Available: https://arxiv.org/abs/1606.01540

[50] N. Koenig and A. Howard, “Design and use paradigms for gazebo, an open-source multi-robot simulator,” in Proc. IEEE/RSJ Int. Conf. Intell. Robots Syst. (IROS), vol. 3, Sep./Oct. 2004, pp. 2149–2154.

[51] M. Wise, M. Ferguson, D. King, E. Diehr, and D. Dynesci, “Fetch and freight: Standard platforms for service robot applications,” in Proc. Workshop Autonomous Mobile Service Robots, 2016, pp. 1–6.

[52] M. Quigley, B. Gerkey, K. Conley, J. Faust, T. Foote, J. Leibs, E. Berger, R. Wheeler, and A. Ng, “ROS: An open-source Robot Operating System,” in Proc. ICRA Workshop Open Source Softw., Kobe, Japan, vol. 3, 2009, p. 5.

[53] D. P. Kingma and J. Ba, “Adam: A method for stochastic optimization,” in Proc. 3rd Int. Conf. Learn. Represent. (ICLR), Y. Bengio and Y. LeCun, Eds. San Diego, CA, USA: ICLR, May 2015.

[54] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, and A. Desmaison, “PyTorch: An imperative style, high-performance deep learning library,” in Advances in Neural Information Processing Systems 32, H. Wallach, H. Larochelle, A. Beygelzimer, F. d’Alché-Buc, E. Fox, and R. Garnett, Eds. Red Hook, NY, USA: Curran Associates, 2019, pp. 8026–8037.

[55] I. Zamora, N. G. Lopez, V. M. Vilches, and A. H. Cordero, “Extending the OpenAI gym for robotics: A toolkit for reinforcement learning using ROS and Gazebo,” 2016, arXiv:1608.05742. [Online]. Available: http://arxiv.org/abs/1608.05742

CHANDRA KUSUMA DEWA received the bachelor’s and master’s (cum laude) degrees in computer science from Universitas Gadjah Mada, Yogyakarta, Indonesia, in 2008 and 2011, respectively. He is currently pursuing the Ph.D. degree with the Toyohashi University of Technology, Toyohashi, Japan. In 2013, he joined the Department of Informatics, Universitas Islam Indonesia, Yogyakarta. His research interests include deep reinforcement learning and robotics.

JUN MIURA (Member, IEEE) received the B.Eng. degree in mechanical engineering, and the M.Eng. and Dr.Eng. degrees in information engineering from The University of Tokyo, Tokyo, Japan, in 1984, 1986, and 1989, respectively. In 1989, he joined the Department of Computer-Controlled Mechanical Systems, Osaka University, Suita, Japan. Since April 2007, he has been a Professor with the Department of Computer Science and Engineering, Toyohashi University of Technology, Toyohashi, Japan. From March 1994 to February 1995, he was a Visiting Scientist with the Computer Science Department, Carnegie Mellon University, Pittsburgh, PA. He has published over 220 articles in international journal and conferences in the areas of intelligent robotics, mobile service robots, robot vision, and artificial intelligence. He received several awards, including the Best Paper Award from the Robotics Society of Japan, in 1997, the Best Paper Award Finalist at ICRA-1995, and the Best Service Robotics Paper Award Finalist at ICRA-2013.

* * *

191838 VOLUME 8, 2020