2021 International Conference on
Artificial Intelligence (ICAI)
Islamabad, Pakistan, April 05-07, 2021

Motion Planning for a Snake Robot using Double Deep Q-Learning

Semab Neimat Khan*[1], Tallat Mahmood*[1], Syed Izzat Ullah[1], Khawar Ali[1], Anayat Ullah[1,2]
[1]Control, Automotive and Robotics Lab, National Centre of Robotics and Automation,
Rawalpindi, Pakistan.
[2]Department of Electronic Engineering, Balochistan University of IT, Engineering and Management Sciences,
Quetta, Pakistan.
Email:semabbuzdar@gmail.com, tallat.mahmd@gmail.com, syedizzatullah@gmail.com, khawar.mohazzam@gmail.com,
anayatullah.baloch@googlemail.com

2021 International Conference on Artificial Intelligence (ICAI) | 978-1-6654-3293-1/20/$31.00 ©2021 IEEE | DOI: 10.1109/ICAI52203.2021.9445200

***Abstract*—Motion planning for a snake robot in an unknown complex environment is a long-standing research problem because of the complex control of the modular mechanism. We propose deep reinforcement learning-based novel framework for motion planning. In this model-free framework, we propose a double deep Q-learning-based technique to learn the optimal policy for reaching the goal point from a random start point; in a minimum number of steps in various unknown environments. In this approach, the agent learns to minimize the distance between the current and goal positions by aligning its yaw angle to the goal points through controlling multiple locomotive gaits. For experimental evaluation, we trained and tested the model in obstacle-free terrains. For training, we selected the model on the mud-terrain and tested for 50 episodes on five different terrains concrete, default, metallic, mud, and wooden. From simulation results, we observe the learned-optimal policy shows promising results for all unknown environments with a performance efficiency of 100% for all terrains except the wooden-terrain where it fails for only one episode and achieves 98% efficiency.**

***Index Terms*—Double deep Q learning, Experience replay memory, Model free, Motion planning, Off-policy, Snake robot.**

I. Introduction

Increasingly robotics is being applied to real-world problems, be it industrial, defense, or civilian applications. The robots help people to increase efficiency as well as the quality of work. Various types of robots have been developed in the last few decades, but recently bio-inspired robots [1] have caught great attention of the research community done for their agility and unique physical capabilities that allows them to navigate and transverse varied environments. Among the bio-inspired robots, the limbless robots (a snake robot) have a diverse range of applications because of their high agility and adaptability. The idea of a snake robot was first introduced by Prof. Hirose [2] and its mechanics were first described in [3].

The snake robots are designed as modular mechanisms to achieve the agility and adaptability of their biological counterparts [4]. The high flexible modular joints with many Degrees of Freedom (DoF) assist them to change their shape and navigate into a highly cluttered environment. However, this modularity comes with its challenges. The control of such a highly maneuverable robot requires controlling a lot of joint actuators and considering their physical constraints for motion control.

Over the past many years a lot of work has been done in snake locomotion (gait design) focusing on low-level control inputs to individual joints using sinusoid-based methods [2], dynamics-based methods [5], and central pattern generator based methods [6]. These gaits have rhythmic functions that change a snake robot's shape in the form of a wave propagating along its body. An approach presented in [7], glued high-level motion planner and low-level control for a snake robot and accomplished motion planning using conventional algorithms. Their algorithm lacked results for different terrains and environmental adaptability. The complex control task includes the internal regulation of body joints and external interaction with the ground. Therefore, the model-based methods usually fail to control the robots adaptively in a challenging environment [8].

Hence, there exists a need for motion planning algorithms that work independently of the underlying robot's physical nuances and are robust to environmental changes. Some recent work using Reinforcement Learning (RL) for path planning [9] & [10] shows promising results as adaptive motion planners in robotics. An object tracking using deep-RL has been presented in [11] and accomplish robust results in a dynamic environment.

Therefore, RL-framework provides a direction, and we propose a novel deep-RL-based algorithm for the motion planning of a snake robot. In the proposed technique, we use a double Deep Q-network (double-DQN) for motion planning. The double-DQN is a model-free RL-algorithm, which reduces the overestimation and divergence issues of Q-network [12] & [13], and achieves promising results on various tasks [14].

The main objective is to reach the goal point in a minimum number of steps. If the direction of the snakehead is toward the goal point, then it finds the shortest distance. Therefore, we consider the respective angle between the heading direction of the snake and the goal point as state-space. To improve the efficiency of the RL framework, we significantly reduce the state space, which helps in the convergence of our RL-based controller, and is evident from the experimental results. Based

* Equal contribution of the authors.

on the respective angle, the artificial agent (double deep Q-learning) learns the optimal policy to select the specified gait to reach the goal point in a minimum number of steps. To verify the efficiency and robustness of the proposed algorithm, we trained our model in mud terrain and tested it on five different unknown terrains of the CoppeliaSim simulator. In a series of simulated experiments, we demonstrated the effectiveness of our proposed controller in varied environmental conditions and its resilience based on mean-steps and performance efficiency. From the experimental results, it is evident that the agent reaches the goal point with a performance efficiency of 100% (tested for 50 episodes) in concrete, default, metallic, and mud-terrain using the learned policy. While it fails on one episode of the wooden terrain, where the efficiency is 98% .

II. Background

A. Reinforcement Learning Framework

A Reinforcement Learning (RL) system learns how an agent achieves its goal by trial-and-error interactions with the environment [12]. The agent based on the observations, interacts with the environment by performing an action. After the action is performed in a given state and a new state is achieved, the RL agent receives some reward; a numerical value $R \in \mathbb{R}$. The main objective of the RL agent is to learn the actions, which will maximize the future cumulative rewards (long-term expected return) when starting from some initial state and proceeding to a terminal state. Hence, the agent only explores the actions which have the highest future rewards, without expressing which actions to take.

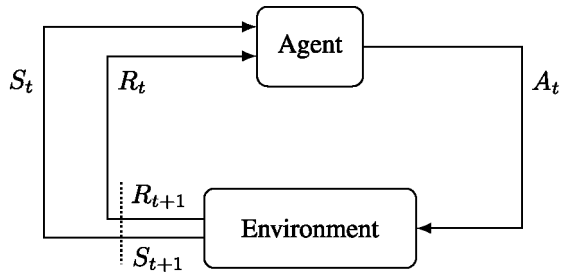


Fig. 1: Illustration of agent-environment interaction in RL framework. Where S_t represents the state of an agent at time step t, based on observations selects an action A_t and receives a reward R_{t+1} after transiting to a new state S_{t+1}.

A general RL problem is formulated as a discrete time stochastic process in which an agent interacts with its environment at each time step; $t = 0, 1, 2, \ldots$. The agent at time step t at the environment's state $S_t \in \mathcal{S}$ ($\mathcal{S}$ is a finite set of states) and based on observations selects an action $A_t \in \mathcal{A}(s)$ ($\mathcal{A}(s)$ a finite set of actions), as illustrated in Fig. 1. After one time step, as a consequence of its action, the agent receives a numerical reward, $R_{t+1} \in \mathcal{R} \subset \mathbb{R}$ and, transitions into a new state S_{t+1}. Where S_t and A_t are the random variables having well defined discrete probability distributions. For a specific value of these random variables $s' \in \mathcal{S}$ and $a' \in \mathcal{A}$, there is a probability of those values occurring at time t. If the sequence of rewards received after time step t is denoted as $R_{t+1}, R_{t+2}, R_{t+3}, \ldots$, then expected discounted return denoted G_t is defined as some specific function of the rewards sequences as:

$$\begin{aligned} G_t \doteq & R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \ldots \\ = & R_{t+1} + \gamma G_{t+1}, \end{aligned} \tag{1}$$

where $\gamma \in [0, 1]$ is a discount rate or factor, which defines the short and farsightedness of the agent. While an equal sign with a dot ($\doteq$) represents equivalent by definition.

One of the novel technique in RL is Temporal Difference (TD) learning [12]. TD methods learn directly from raw experience without having the environment's model, update estimations based on different learned estimates, without waiting for an outcome [15]. TD and optimal control [16] are combined in Q-learning [17]. The optimal control finds a mapping that prescribes actions based on measured environmental states to optimizes some long-term rewards. For estimation of optimal value, Q-learning algorithm builds the action-value function, a primary part of reinforcement learning [17].

B. Q-Learning

A sequential decision problem can be solved by learning the estimated optimal values of each action, defined in terms of the expected sum of future reward G_t. The value of an action a in a state s ($Q_\pi(s, a)$) by following the policy π is defined as:

$$Q_\pi(s, a) \doteq \mathbb{E}_\pi[G_t | S_t = s, A_t = a], \quad \forall\, s \in \mathcal{S}, a \in \mathcal{A}. \tag{2}$$

The optimal policy can be computed from the optimal values ($Q_*(s, a) = \max_\pi Q_\pi(s, a)$) by selecting the highest value action in each state.

The optimal action values (estimates) can be learned by a Q-learning algorithm [17], a form of off-policy TD learning [15]. The off-policy learning evaluates and improves one policy and selects an action based on another policy. For many complex robotic problems, the computation of action values for all states is a complicated task. Therefore, a parametrized value function $Q(s, a; \boldsymbol{w})$ can be incorporated, where $\boldsymbol{w}$ indicates the parameters. The Q-learning update parameters, after taking an action A_t in-state S_t and transition to a new state S_{t+1} while receiving a reward R_{t+1}, is:

$$\begin{aligned} \boldsymbol{w}_{t+1} = \boldsymbol{w}_t + \alpha \big(& R_{t+1} + \gamma \max_a Q(S_{t+1}, a; \boldsymbol{w}_t) \\ & - Q(S_t, A_t; \boldsymbol{w}_t) \big) \nabla_{\boldsymbol{w}_t} Q(S_t, A_t; \boldsymbol{w}_t), \end{aligned} \tag{3}$$

where α represents a learning rate. While the target is represented as:

$$Y_t^Q \doteq R_{t+1} + \gamma \max_a Q(S_{t+1}, a; \boldsymbol{w}_t). \tag{4}$$

The max operator in Eq. 4 expresses the Q-learning algorithm chose the greedy values, which results in overoptimistic value estimation.

C. Double Q-Learning

To avoid the overestimation problem of Q-learning, double Q-learning algorithm is presented in [18]. It decomposes the max operation of the target into action selection and action

evaluation. In double Q-learning algorithm, two value estimation functions learn from experience by randomly updating one of the value estimation functions. Therefore, it contains two sets of weights, $\boldsymbol{w}_t$ and $\boldsymbol{w'}_t$. For each update, one set of weights compute the greedy policy while the other compute its value. The target of double Q-learning is presented as:

$$Y_t^{\text{DoubleQ}} \doteq R_{t+1} + \gamma Q(S_{t+1}, \arg\max_a Q(S_{t+1}, a; \boldsymbol{w}_t); \boldsymbol{w'}_t) \tag{5}$$

For Eq. 5, one can observe that action is selected, in the arg max, from online weights $\boldsymbol{w}_t$. Which represents the Q-learning estimates the value from a greedy policy. While the double Q-learning uses a second set of weights $\boldsymbol{w'}_t$ to evaluate the value of this greedy policy. However the second set of weights are updated symmetrically by switching the roles of $\boldsymbol{w}_t$ and $\boldsymbol{w'}_t$. The double Q-learning avoids the overestimation problem, but combining model-free RL algorithms (Q-learning) with non-linear function approximation may cause the Q-learning algorithm to diverge, because of the correlation between samples and non-stationary targets [13]. To solve these issues the Deep Q-Network (DQN) has been presented in [14].

D. Deep Q-Networks

With the recent improvement in deep neural networks, a novel artificial agent termed DQN has been presented in [14]. DQN is a multi-layered neural network, which generates a vector of action values $Q(s, \cdot; \boldsymbol{w})$ for a given state s, while $\boldsymbol{w}$ represents the parameter or weights of the network. The neural network maps n-dimensional state space $\mathbb{R}^n$ to m-dimensional action space $\mathbb{R}^m$. To resolve the divergence problem of Q-learning, the DQN uses a fixed target network along with experience replay. The experience replay addresses the issue of correlation. To improve the stability issue, for multiple updates (iterations), fixed target parameters or weights $\boldsymbol{w}_t^{tar}$ are used in the target value calculation, the weights are being updated with $\boldsymbol{w}_t$. The target of DQN is then:

$$Y_t^{DQN} \doteq R_{t+1} + \gamma \max_a Q(S_{t+1}, a; \boldsymbol{w}_t^{tar}). \tag{6}$$

To integrate the experience replay, the observed transitions are stored in the replay memory for some time and the network weights are updated by uniformly sampled values from this memory. From Eq. 6, one can observe the max operator makes the deep Q-learning algorithms to select overestimated values.

E. Double-DQN

Double Q-learning algorithm reduces the overestimation problem while the DQN controls the divergence issue. Therefore, by combining both Double Q-learning and DQN, a new algorithm has been presented in [19], referred to as Double-DQN. The double-DQN uses two identical neural networks, the online and the target network. The double-DQN evaluates the greedy policy by an online network while using the target network of DQN to estimate its value. The target of double-DQN is presented as:

$$Y_t^{\text{DoubleDQN}} \doteq R_{t+1} + \gamma Q(S_{t+1}, \arg\max_a Q(S_{t+1}, a; \boldsymbol{w}_t); \boldsymbol{w}_t^{tar}) \tag{7}$$

In contrast to DQN, the Double DQN has a target network with the parameter $\boldsymbol{w}_t^{tar}$, which evaluates the quality of the actions. The Q-network computes the greedy policy with parameter $\boldsymbol{w}_t$. In comparison to the Double Q-learning (Eq. 5), the weights of the second network $\boldsymbol{w'}_t$ are replaced with the parameters of the target network $\boldsymbol{w}_t^{tar}$ for the evaluation of the current greedy policy. The update to the target network is the same as for the DQN and contains a periodic copy of the online network.

III. Proposed Methodology

We propose an RL-based scheme for motion planning of a snake robot (agent) using the Double-DQN. The main objective of the agent is to reach the goal point (an indoor plant in our case) from its current location, using various locomotive gaits, in minimum time steps. The goal point $\boldsymbol{Q}$ and current location $\boldsymbol{P}$ is shown as a big green dot and a big red dot, respectively, in Fig. 2. To achieve this objective,

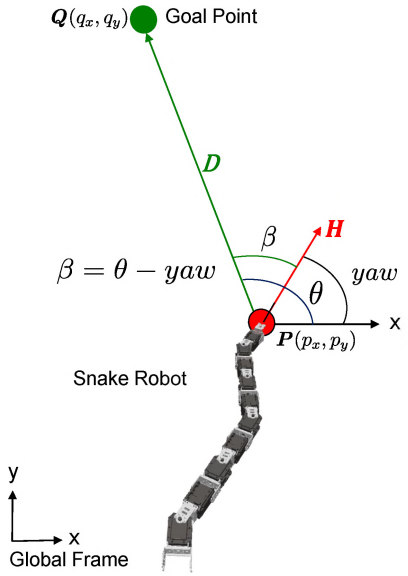


Fig. 2: The points $\boldsymbol{P}(p_x,p_y)$ and $\boldsymbol{Q}(q_x,q_y)$ define the current position of the snake robot and goal respectively. The angle β is the respective angle between the snake's current heading vector $\boldsymbol{H}$ and optimal heading vector $\boldsymbol{D}$. The θ represents the angle between vector $\boldsymbol{D}$ and reference x-axis.

we use the Double-DQN algorithm presented in [19] using a linear ϵ-greedy policy, in which ϵ has value in range $[0, 1]$ and decays linearly from ϵ_i to ϵ_f. At each time step, we generate a random number between $(0, 1)$ from a uniform distribution. If this random number is less than the current value of ϵ, the agent takes a random action a from the available action space $\mathcal{A}$. Otherwise, we pass the current state $s \in \mathcal{S}$ to the online network, and the agent takes the action a having the maximum state-action value $Q(s, a)$. In response to action a, the agent moves to a new state s' and receives a reward R_{t+1} from

the environment. At each time step, the tuple (s, a, R_{t+1}, s'), referred to as a transition, and a *Done* flag is stored in the Experience Replay Memory(ERM). Where the *Done* flag is a Boolean variable, which indicates the termination of an episode. At each time step t, a batch of transitions is taken from the ERM, and weights of the online network of double-DQN are updated using Eq. 3. So the action policy π is updated to π', as shown in Fig. 3. In the proposed scheme,

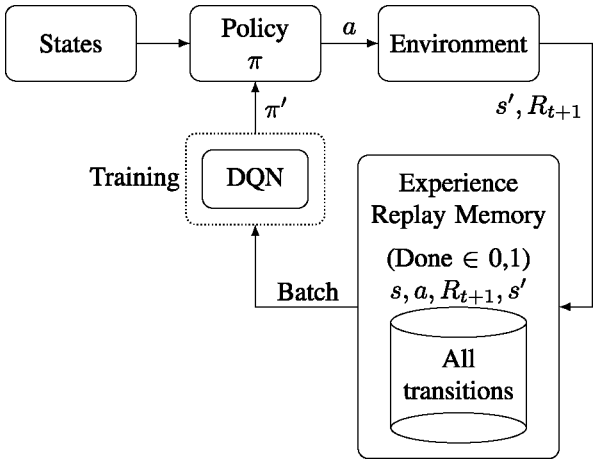


Fig. 3: Illustration of double DQN training process. The agent observes the current state s, takes an action a using policy π, transits to the next state s', receives a reward R_{t+1} from the environment and *Done* flag stores episodes's completion information. All these transitions are logged in the ERM. At each step a batch of transitions is used, to update the agent's policy π.

we use β to formulate the states of the agent. Where β is the angle between the snake heading vector $\boldsymbol{H}$ (red vector) and vector $\boldsymbol{D}$ (green vector), as shown in Fig. 2, and computed as:

$$\beta = \theta - yaw \tag{8}$$

Here yaw is the angle between the vector $\boldsymbol{H}$ and x-axis provided by the simulator. Whereas, θ is the angle of vector $\boldsymbol{D}$ with respect to the x-axis and computed as:

$$\theta = tan^{-1}\frac{q_y - p_y}{q_x - p_x}. \tag{9}$$

In reinforcement learning, a low dimensional state space $\mathcal{S}$ is computationally robust compared to a high dimensional state-space. As in our case, the respective angle β has infinite values in the given range $[-180° \sim 180°]$, which leads to high complexity. Moreover, the locomotive gaits can not respond to minute changes (less than 1°) in respective angles. To solve this problem, we map the value of the snake heading direction using the continuous values of angle β to integer i using:

$$i = f(\beta) = \left\lfloor \frac{\beta}{3} + 60 \right\rfloor, \quad \text{where} \quad -180 \leq \beta \leq 180 \in \mathbb{R} \quad \text{and} \quad 0 \leq i \leq 120 \in \mathbb{Z}^{0+} \tag{10}$$

Furthermore, we assign a one dimensional vector $\boldsymbol{V^i}$ for every integer i of dimension 1×121, as shown in Fig. 4. The

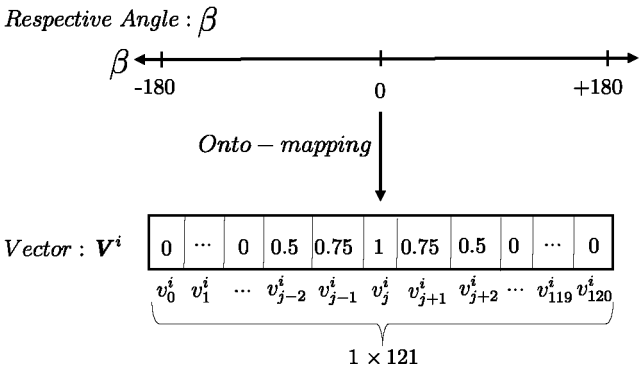


Fig. 4: The illustration of onto-mapping of respective angle β to the Vector $\boldsymbol{V^i}$. The domain of this mapping function is of infinite dimension $(-180 \leq \beta \leq +180 \in \mathbb{R})$ while the range is finite (121 vectors $\boldsymbol{V^i}$, each of dimension $[1 \times 121]$). The v_j^i represents the j^{th} component of the vector $\boldsymbol{V^i}$.

component of vector $\boldsymbol{V^i}$ can be represented as v_j^i where j represents the index of that component $(0 \leq j \leq 120 \in \mathbb{Z}^{0+})$ of vector $\boldsymbol{V^i}$. The values of these components are computed as:

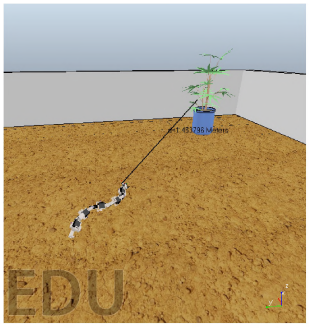
$$v_j^i = \begin{cases} 1, & \text{if } \ j = i \\ 0.75, & \text{if } \ j = i \pm 1 \\ 0.5, & \text{if } \ j = i \pm 2 \\ 0, & \text{otherwise} \end{cases} \tag{11}$$

After computing the vector $\boldsymbol{V^i}$, we produce a $2D$ array of dimension 6×121 by replicating the vector $\boldsymbol{V^i}$ 6 times for a respective angle β of a single state. The set of all these $2D$ arrays builds our state-space $\mathcal{S}$, so the agent has 121 discrete states.

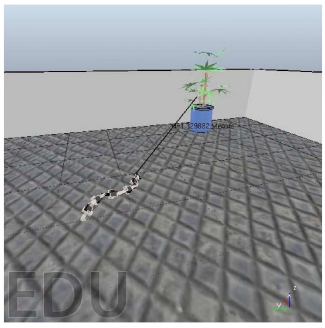
The objective of our agent is to minimize the angle β by readjusting its yaw angle and trying to reach the goal point using various locomotive gaits, in a minimum number of steps as shown in Fig. 2. To simplify the complexity of motion, we use only three locomotive gaits of the snake robot as action-space $\mathcal{A}$: rectilinear, right-serpentine, and left-serpentine [3]. The target of the agent is to learn the optimal action for locomotion to reach the goal point, with the highest rewards received after each action.

After an action a is performed the agent moves to a new location, therefore the distance from the goal location changes which in turn changes the angle β. If the agent receives a reward or penalty based on β, the agent takes the action to remain in the direction of the goal location. After the agent's heading aligns with the direction of the goal point, the agent only actuates the forward locomotive gait, to reach the goal point by following the shortest possible path. Hence, we compute the reward function R_t based on respective angle β defined as:

$$R_t = \begin{cases} 100, & \text{if terminal state} \\ -1 - \left|\frac{\beta}{20}\right| & \text{otherwise} \end{cases} \tag{12}$$



(a) CoppeliaSim environment in which snake robot and goal point can be seen on mud terrain.



(b) CoppeliaSim environment in which snake robot and goal point can be seen on metallic terrain.

Fig. 5: CoppeliaSim environment in which snake robot and goal point can be seen on different terrains.

The vicinity of one meter to the goal point is considered as the terminal state of the episode, and the agent receives a positive reward of 100. For all other states, it is penalized based on the respective angle β. It receives a minimum penalty when the value of β is a small and high penalty for higher value of β. From these reward values, the agent evaluates the actions and responds to the environment accordingly.

IV. Experimental Evaluation

We used a robot simulator "CoppeliaSim $V-4.1$" for our experimental evaluation (for more details see [20]). CoppeliaSim is a well-known physics simulator among the robotics community which provides flexibility to create robots of various kinds by importing pre-designed 3D-CAD models. For our experimental evaluation, we used an obstacle-free environment of 4×6 meters. It comprises of a snake robot as the agent and an indoor plant as a goal point. The hyper-redundant structure of our snake robot consists of ten modular joints, which are segregated into three groups; head, body, and tail. We performed the training on the mud-terrain to mimic the real-world environment, as shown in Fig. 5a. We used two identical neural networks, online network and target network, for double-DQN. The neural network consists of a convolution layer with 32 kernels of size 2×2, one flatten layer of size 1×5760, one fully connected layer of 64 neurons, and an output layer as shown in Fig. 6. The size of the output layer is equal to the number of actions available (three in our case).

Moreover, we used stride of 2×2 with "relu" as the activation function and "Adam" as the optimizer. At the output layer, we used a linear activation function, which gives the estimated $Q(s, a)$ value for each state-action pair. We used a discount factor (γ) of 0.98 and a learning rate (α) of 0.00025 for target policy Eq. 7 and weights are updated as mentioned in Eq. 3. We used linear ϵ-greedy policy which is updated after each step, with an initial value $\epsilon_i = 1$ and final value $\epsilon_f = 0.1$. The target network was updated after 25 episodes, while an episode is the maximum number of steps allowed to reach the goal point. For experimental evaluation, we selected random samples in batches of 64, from ERM, with maximum samples of $30,000$, and minimum samples of 500. In this

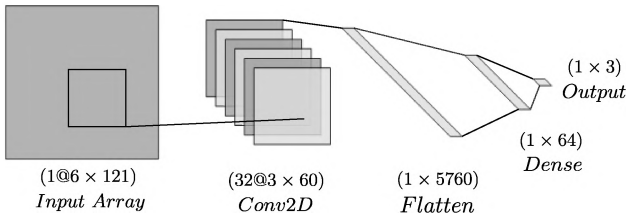


Fig. 6: Illustration of the network architecture which consists of input layer, hidden layers and an output layer of dimensions $(1@6 \times 121)$, $(32@3 \times 60,\ 1 \times 5760,\ 1 \times 64)$ and (1×3) respectively.

experiment, we used 100 steps per episode. All the hyper-parameters of Double-DQN are shown in Tab. I. We developed

Parameter	Value
Activation function for hidden layers	relu
Activation function for output layer	linear
Optimizer	Adam
Discount factor (γ)	0.98
Learning rate (α)	0.00025
Initial epsilon (ϵ_i)	1
Final epsilon (ϵ_f)	0.1
Update epsilon (in steps)	1
Target network update frequency	25
Minimum replay memory size	500
Replay memory size	30000
Batch size	64
Maximum steps per episode	100

TABLE I: The hyperparameters used to train the double-DQN.

the neural network for training the Double-DQN in Python using TensorFlow and Keras as the backend framework. The training was carried out on an Intel Core i5-4690 CPU (with 4 processing cores each running at 3.5 GHz) and 4GB RAM. The training process took about 22 hours to complete, details can be found in the following section.

A. Training Results

We trained the robot using CoppeliaSim in mud-terrain with the parameters presented in Tab. I. The agent improved its action policy as the training proceeded, depicted in Fig. 7. The dark blue curve in Fig. 7 represents the rolling mean of the step rewards while the spread around it is the standard deviation (Std) of the step rewards. As ϵ decays, the mean and Std of step reward increase, since the agent is nearing the goal point. After $20,000$ training steps, shown as a red dotted line, the value of ϵ reaches 0.1 and the mean reward curve becomes stable. This stable reward indicates that the agent has reached the goal point. The learning process of the agent is shown in Fig. 8. The blue and green curves indicate the rolling mean reward per episode and steps per episode, respectively. The negative slope of the green curve shows that, on average, the agent reaches the goal point in fewer steps as the number of episodes increases. The initial positive slope of the blue curve expresses the increase of reward per episode.

During the initial phase of the training, the number of steps per episode is high $(90-100)$ leading to lower episodic reward

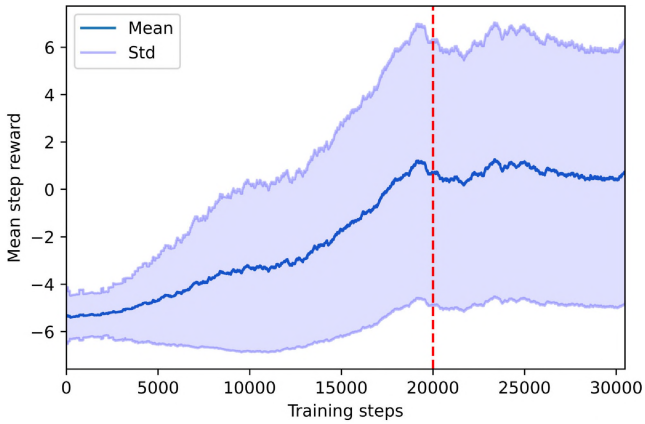


Fig. 7: The blue curve represents the rolling mean step reward and spread around is the rolling standard deviation during training. The dotted red line indicates the point where ϵ became 0.1.

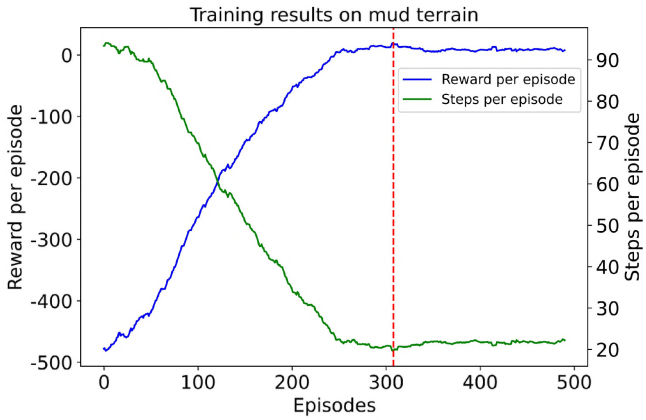


Fig. 8: The blue curve represents the rolling mean rewards per episode while training whereas, the green curve represents the rolling mean of steps taken by the agent in each episode.

(−500 to −600). This indicates that initially the actions are not optimized and the agent is not intelligent enough to reach the goal. After nearly 308 training episodes and 20,000 steps, the agent sufficiently improves the action policy. At that point, the decay in epsilon is stopped so that the agent can keep on exploring the environment. The training process is continued so the agent can optimize its policy further.

During training, the highest episodic reward of 72.65 was observed at 660 per episode. We stopped the training at 662 episodes and 32967 steps. At the end of the training phase, the mean steps per episode are between 20 to 30, and the higher mean episodic reward is between 55 to 65. So, the proposed method robustly trains the agent to maximize the commutative reward and minimize the number of steps per episode.

B. Testing Results

We used multiple checkpoints to log the training rewards, steps per episode, *Done* flag, and episode counts for the target network. The highest future cumulative reward G_t implies

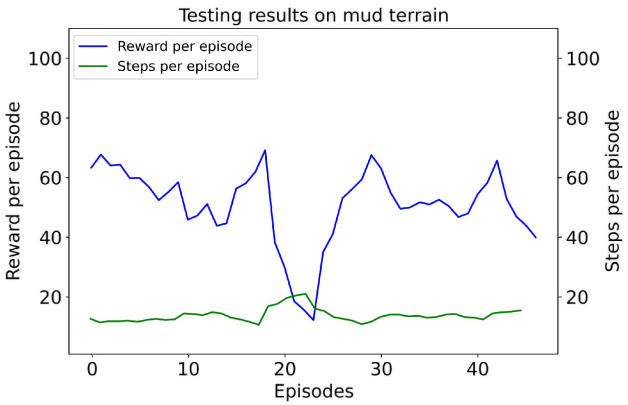


Fig. 9: This illustrates the testing learned policy on mud terrain, the blue curve represents the rolling mean rewards per episode whereas, the green curve represents the rolling mean of steps taken by the agent in each episode.

that the particular episode carries the best weights among the previous ones. We tested the learned policy for 50 episodes, as shown in Fig. 9. During testing of mud-terrain, the agent reached the goal point successfully with a high cumulative reward in less than 80 steps. It can be assessed from Fig. 8, that the episodic reward during the initial training phase is approximately −500. However, after learning the optimal policy in mud-terrain, the snake robot reached a goal point in fewer steps and received a high cumulative reward when tested on the same terrain. This learned policy was also evaluated on various unknown environments: concrete, metallic, wooden, and default terrain available in the CoppeliaSim simulator. The test results in terms of steps per episode for various terrain is shown in Fig. 10. From the graph one can observe during testing, the agent reaches the goal point in all episodes except one, wood-terrain, where the agent could not reach the target. This result shows the efficiency and robustness of the proposed scheme to reach the goal point. Since, the agent was trained in a different environment and tested in a different environment. So one can argue that the proposed scheme shows promising results in an unknown environment.

We categorize the episodes into successful and unsuccessful.

Terrains	**Mean reward**	**Mean steps**	**Total time (minutes)**	**Efficiency %**
Concrete	-3.88	24.55	45.89	100
Default	13.37	21.22	39.36	100
Metallic	7.95	23.00	43.34	100
Mud	38.74	16.92	32.46	100
Wooden	9.67	21.24	39.46	98

TABLE II: Illustration of testing results of learned policy on various terrains. On mud terrain, the agent reaches the goal location in a minimum number of steps and time, with maximum reward. The learned policy shows 100% efficiency on different terrains except for wooden terrain, where 98% efficiency is achieved.

A successful episode is the one in which the agent reached the goal point, while the efficiency is the ratio of the total number of successful episodes to all episodes. For training,

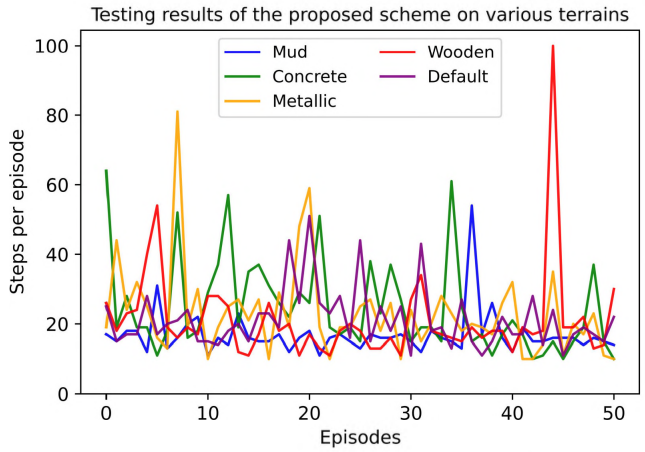


Fig. 10: Illustration of the steps per episode while testing the learned policy on various terrains. Most of the time the agent reaches the goal position within 40 steps. Just in one episode of wooden terrain, the agent could not reach the goal position.

the efficiency was calculated after every 50 episode, while during the testing the efficiency was computed after every 5 episodes. In Tab. II, we show the results of the proposed schemes tested on various unknown environments in terms of mean reward, mean steps, total time, and performance efficiency. From Tab. II, one can observe the agent converges more robustly in terms of mean step, total time, efficiency, and higher rewards when tested in mud-terrain. The test results of various unknown environments is also promising, and achieve test accuracy of 100% for concrete, metallic and default-terrain, and 98% accuracy in wooden-terrain. From Tab. II, we can argue that the proposed scheme has high efficiency in various unknown environments.

V. Conclusion

Motion planning for a snake robot (modular mechanisms) is a challenging task. Because of complex control tasks, the model-based methods are not robust to control the robot adaptively in a challenging environment. We propose a double-DQN based scheme to optimize the gait selection of a snake robot; to reach the random goal point in an unknown environment in a minimum number of steps. When it is trained on mud-terrain and tested on various terrains like mud, metallic, default, and wooden-terrain in CoppeliaSim, the proposed learning-based scheme shows promising performance efficiency on various unknown environments without changing the parameters. In future, we are planning to work in a complex dynamic environment, like adding static, dynamic and vision-based object detection. Furthermore, the proposed scheme will be tested in an experimental setup to evaluate its performance.

Acknowledgment

This research is conducted at Control Automotive and Robotics Lab (CARL-BUITEMS), funded by National Center of Robotics and Automation (NCRA) with the collaboration of Higher Education Commission (HEC) of Pakistan.

References

[1] P. Dario, "Biorobotics," *Journal of the Robotics Society of Japan*, vol. 23, no. 5, pp. 552–554, 2005.

[2] T. Owen, "Biologically inspired robots: Snake-like locomotors and manipulators by shigeo hirose," *Robotica*, vol. 12, no. 3, pp. 282–282, 1993.

[3] J. Gray, "The mechanism of locomotion in snakes," *Journal of Experimental Biology*, vol. 23, no. 2, pp. 101–120, 1946.

[4] I. D. Walker and M. W. Hannan, "A novel elephant's trunk robot," in *IEEE/ASME International Conference on Advanced Intelligent Mechatronics*, Atlanta, GA, USA, September 1999, pp. 410–415.

[5] G. S. Miller, "The motion dynamics of snakes and worms," in *15th annual conference on Computer graphics and interactive techniques*, New York, NY, United States, August 1988, pp. 169–173.

[6] Z. Bing, L. Cheng, G. Chen, F. Röhrbein, K. Huang, and A. Knoll, "Towards autonomous locomotion: Cpg-based control of smooth 3d slithering gait transition of a snake-like robot," *Bioinspiration & biomimetics*, vol. 12, no. 3, p. 035001, 2017.

[7] R. L. Hatton, R. A. Knepper, H. Choset, D. Rollinson, C. Gong, and E. Galceran, "Snakes on a plan: Toward combining planning and control," in *2013 IEEE International Conference on Robotics and Automation*. IEEE, 2013, pp. 5174–5181.

[8] Z. Bing, C. Lemke, Z. Jiang, K. Huang, and A. Knoll, "Energy-efficient slithering gait exploration for a snake-like robot based on reinforcement learning," *28th International Joint Conference on Artificial Intelligence*, August 2019.

[9] F. Morbidi and G. L. Mariottini, "Active target tracking and cooperative localization for teams of aerial vehicles," *IEEE transactions on control systems technology*, vol. 21, no. 5, pp. 1694–1707, 2012.

[10] W. Luo, P. Sun, F. Zhong, W. Liu, T. Zhang, and Y. Wang, "End-to-end active object tracking and its real-world deployment via reinforcement learning," *IEEE transactions on pattern analysis and machine intelligence*, vol. 42, no. 6, pp. 1317–1332, 2019.

[11] Z. Bing, C. Lemke, F. O. Morin, Z. Jiang, L. Cheng, K. Huang, and A. Knoll, "Perception-action coupling target tracking control for a snake robot via reinforcement learning," *Frontiers in Neurorobotics*, vol. 14, p. 79, 2020.

[12] R. S. Sutton and A. G. Barto, *Introduction to reinforcement learning*. MIT press Cambridge, Massachusets, USA, 2018, vol. 2.

[13] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. Riedmiller, "Playing atari with deep reinforcement learning," in *NIPS Deep Learning Workshop*, Lake Tahoe, USA, December 2013.

[14] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski *et al.*, "Human-level control through deep reinforcement learning," *Nature*, vol. 518, no. 7540, pp. 529–533, 2015.

[15] R. S. Sutton, "Learning to predict by the methods of temporal differences," *Machine learning*, vol. 3, no. 1, pp. 9–44, 1988.

[16] J. K. Williams, "Reinforcement learning of optimal controls," in *Artificial intelligence methods in the environmental sciences*. Springer, 2009, pp. 297–327.

[17] C. J. C. H. Watkins, "Learning from delayed rewards," Ph.D. dissertation, King's College, Cambridge, United Kingdom, 1989.

[18] H. Hasselt, "Double q-learning," *Advances in neural information processing systems*, vol. 23, pp. 2613–2621, 2010.

[19] H. Van Hasselt, A. Guez, and D. Silver, "Deep reinforcement learning with double q-learning," *AAAI Conference on Artificial Intelligence*, 2015.

[20] E. Rohmer, S. P. Singh, and M. Freese, "Coppeliasim (formerly v-rep): a versatile and scale-able robot simulation framework," in *Proc. of The International Conference on Intelligent Robots and Systems (IROS)*, Tokyo, Japan, November 2013.