Using Reinforcement learning for Control of Microrobots through Local Potential Fields

Kumaraguru Sivasankaran[1], Shubhankar Gupta[1], and Benjamin V Johnson[2]
[1]*School of Aeronautics and Astronautics,* [2]*Mechanical Engineering Purdue University, West Lafayette, IN, 47907*

In this work, we explore the possibility of applying various learning algorithms for controlling micro-robots. We start with simple deterministic cases and use dynamic programming based approaches to find solutions to optimal path problem. Due to uncertainties, however, we then model system as a stochastic environment and used reinforcement learning to solve the problem. We found SARSA algorithm as dependable reinforcement learning as it was easy to implement. Further, this work could be possibly the first to investigate on applying Deep reinforcement learning for controlling micro-robots. The initial results show promising trend and motivate to investigate further. The success we got with hardware level implementation show the maturity towards the understanding of the problem of controlling micro-robots.

I. Introduction and Problem Statement

A reinforcement learning agent interacts with its environment in discrete time steps. At each time t, the agent receives an observation which typically includes the reward. It then chooses an action from the set of available actions, which is subsequently sent to the environment. The environment moves to a new state and the reward associated with the transition is determined. The goal of a reinforcement learning agent is to collect as much reward as possible. The agent can (possibly randomly) choose any action as a function of the history. In the field of Control theory, reinforcement learning is also referred to as Approximate Dynamic programming relating this project to our current course work. With this background, we introduce our problem of controlling micro-robots.

Traditionally, micro-robot manipulation using external magnetic fields is accomplished in two steps: developing a path planning algorithm (such as A* algorithm) and then, motion control using hand crafted control update equations. This works fine when the environment is deterministic, robot motions for given control inputs are identified through prior experiments and has less/ zero disturbances. When some of these parameters change under the current setting, one must redevelop the path planning and controller algorithm. This process is quite tedious and inefficient. Further, in the event of more than one robot where the robots may work independently or cooperatively, designing a control algorithm is very challenging. To overcome this, we propose to apply a model-free off-policy Reinforcement learning algorithm similar to Deep Q-learning for developing an end-to-end controller[1].

The micro-robot learning will be based on the algorithm 1 which takes camera image as the state information and actions based on controlling individual coil.

One of the drawbacks of this approach is that, the algorithm requires many training examples and takes long time to converge to the optimal solution. Thus, the experiments may turn out to be costly. To overcome this situation, we plan to develop a simulated environment of the actual problem in Python/MATLAB as first step where the robot will learn the optimal policy. Then, the learned policy is perfected in the actual environment. It would be interesting to see the robustness of the learned policy as there are differences in simulated and actual environment. Based on the initial success, we would like to extend it to multiple robot case with cooperation/ independently controlled. It is important to note that the computer vision and deep learning algorithm for Q-function approximation is quite challenging for real time systems and would be the major contribution of this work. At the end of this work, we would like to present the theory of the algorithm, different variants, convergence proof and results of this experiment. The details of the experimental setup is discussed in Section II.B.

II. System Description

Traditionally, magnetic microrobots are driven through external magnetic coils which generate a global magnetic field in the workspace. However, it is challenging to actuate multiple robots using such coils. To enable actuation of

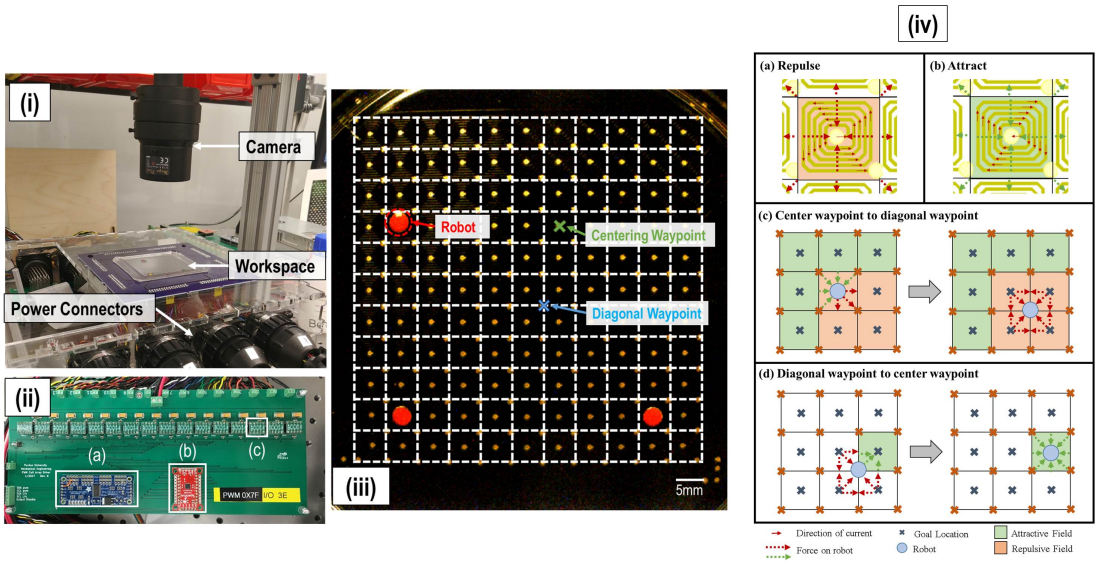


Fig. 1 (i) Workspace: camera, power connectors, and workspace. (ii)(a) PWM driver, (b) I/O Controller, and (c) Motor driver (iii) Workspace view from the camera. The coil grids are shown in white overlay grid. The centering and diagonal waypoints are also displayd. (iv) (a) Local magnetic field actuation. (a) Repulsive force due to clockwise wire current. (b) Attractive force due to anti-clockwise current in the planar coil. (c) Coil actuation states needed to actuate a microrobot to a diagonal waypoint from a center waypoint; (d) States needed to actuate a microrobot to a center waypoint from a diagonal waypoint.

multiple robots, we've custom built a printed circuit board (Osh Park LLC) with a 11×11 grid of spiral coils as shown in the Figure 1. These coils are able to generate magnetic field locally and each coil can be controlled Independently to influence the robots only in its vicinity. Each coil measures 5.04 mm and have 5 turns each, the traces of each coil winding measures 0.36 mm.. The robots we used are 3.125 mm diameter, and 0.5 mm thick N50 disc magnets, magnetized through the thickness.

A. Coil Action Sets

The currents in a spiral coil then will either repulse, force the robot outward, or attract, pull the robot towards it's center, depending on the direction of current flowing through the coil. The net forces on a robot for a fixed direction of current are shown in Fig. 1(iv)(a) and (iv)(b). To define way-points (goal locations) for the robot, it is important that they are at equilibrium points, i.e points at lowest potential. In the previous work[2], it was believed that these equilibrium points only existed at the center of each coil, limiting the way-points for path planning and trajectories to orthogonal moves in the workspace. Here we have identified an additional equilibrium point at the corner positions of each coil (diagonal positions in the workspace). The robot remains in the center of the coil if the coil "attracts" the robot, and it remains in the diagonal (corner) position if all the surrounding coils "repel" the robot. However, to reliably move from a center way-point to a diagonal way-point, all nine neighboring coils of the robot are needed, as shown in Fig.1(iv)(c). Since these states can be reached rather reliably, we use them as states.

B. Experimental Setup

The current in each coil can be controlled through four to eight custom control units (Fig. 1(ii)). Each control unit allows for the individual control of the magnitude and direction of current through sixteen coils. The direction of the current is switched using a voltage level shifter I/O SX1509 (Sparkfun Electronics), and the magnitude of the current is controlled by regulating the supply voltage using a PWM driver PCA9685 (Adafruit) into a motor controller DRV8838

(Pololu Corporation) which supplies the current to drive each coil. A set of 16 coils are controlled using a single GPIO and PWM driver board, to which communications are sent over an I^2C interface. The motor driver provides currents up to 1 A to each coil, and the external current supply can supply up to 60 A of current to the system. For a total of 64 coils, we use 4 controller boards and each unit is connected to the workspace using a circular power connector shown in Fig.1(i). For the larger platform, a total of 121 coils were controlled using 8 controller boards. An Arduino Uno microcontroller is used to communicate with each controller board using I2C signals. The microcontroller is connected to a CPU with an Intel®Core™i7-4771 (3.50 GHz) processor and 16 GB RAM. All computations are performed using Matlab® and the commands are sent to the controller boards through the Arduino microcontroller interface.

The robots used for the experiments are 3.125 mm diameter and 0.79 mm thick neodymium disc magnets. These magnets are of grade N52 and magnetized through their thickness. Multiple robots can move independently in this workspace provided they are approximately 15 mm apart at all times. If they get closer than that, they repel/attract each other, affecting the robot's behavior if they are not in their equilibrium states. The motion of the robots are controlled by controlling the direction and magnitude of currents in the coils in the vicinity of the robot, as discussed in Section II.

These robots are placed with their north pole facing out of plane of the board. The robots are allowed to move on a coverslip glass, while immersed in a layer of silicone oil providing sufficient friction free surface. The robots were tracked using color segmentation and localization using MATLAB image processing toolbox. The camera (PointGrey FL3-U3-13E4C-C, Point Grey Research Inc.) was used to capture images of the workspace at 1Hz, which was sufficient for our application. A matlab code analyzes the captured image and instructs the arduino using the COM ports to activate the required coils in the workspace.

III. Dynamic Programming

A. Dynamic Programming

The system we have in hand is inherently discrete due to its equilibrium points and actions. Hence, this problem can be solved quickly using dynamic programming techniques. The dynamic programming problem is setup as a minimization problem of cost to reach the goal location. For this, we consider $x \in \{1, 2, ..., 265\}$ states corresponding to the diagonal and center way-points, $u \in \{1, 2, 3, 4, 5\}$ which denotes the actions that move the robot to four neighbors and another action to stay in the same location. Since its a free final time problem, the end condition of the algorithm is the encounter of the start location state in the minimization problem when solving using backward recursion. The running cost $l(x, u)$ is set as the distance between the goal and the predicted state from a predicted action. The final cost $\phi(x(N))$ is set as inf for all states except the goal state. The dynamic programming problem can hence be represented as

$$V_j(x(j)) = \min_{u(j),...,u(N-1)} \sum_{k=j}^{N-1} l(x(k), u(k)) + g(x(N)) \tag{1}$$

on applying Bellman's Principle of Optimality,

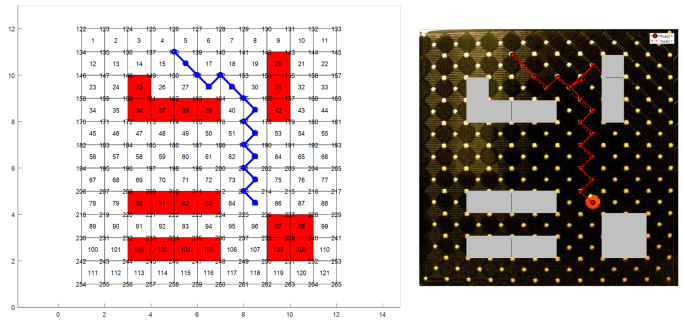
$$V_j(x(j)) = \min_{u(j)}(l(x(j), u(j)) + V_{j+1}(f(x(j), u(j))) \tag{2}$$

MATLAB was used to solve this dynamic programming problem using backward recursion. Obstacles were introduced in the workspace with high cost of movement. Once the starting state was reached during iteration, the final time is noted and then the forward string of states to the goal showing the optimal path is generated by searching for minimization of V from one state to next. A validation simulation and experiment was conducted to observe the effectiveness of the solver, and the results can be seen in Fig. 2.

It is observed that the robot has some deviation from the path. This is attributed to uncertainties in the system. This can be due to coil imperfections, effect of underlying wiring, external influence from other magnets etc. It is therefore important to introduce transition probabilities and model the environment as a stochastic environment.

B. Markov Decision Processes

In view of the uncertainties in the system, the planning problem can be formulated as a Markov Decision Process (MDP) which can formulate the shortest path problem in the stochastic environment. A MDP utilizes information of the



(i) (ii)

Fig. 2 (i) Simulation of the path of the robot from the initial position (138) to goal location (85). (ii) Experiment showing path of the robot in the real system when the doted red lines show the desire way-points and the solid red lines show the actual path of the robot

robot's*(agent)* position*(state)*, *actions* that can be taken that affect the dynamics of the robot, and known rewards for transition between these states. The solution would describe the probability of transition to a state s' when an action a is taken, that would give a reward r. On writing different states and observing rewards, an optimal sequence of actions can be generated. However, due to uncertainties in the system, applying the policy might be different from the final states. However, modeling it as an MDP allows us to extend the model to where the transition and rewards are not known in advance, which will be discussed in Sec. IV

Markov property is defined as the probability of reaching a future state depends only upon the present state, and not the preceding states. MDPs are defined as controlled stochastic processes satisfying the Markov property and assigning reward values to state transitions [3]. They are described by the 5-tuple (S, A, T, p, r).

- $S \rightarrow$ State space
- $A \rightarrow$ Set of all possible actions
- $T \rightarrow$ Time steps for decision to be made
- $p \rightarrow$ State transition probability function
- $r \rightarrow$ Reward function definition on state transitions

The set T here defines the decision epochs which can be finite or infinite, or in our case is defined to correlate with the existence of terminal state(goal location). The transition probability p characterizes the dynamics of the system. $p(s'|s, a)$ represents the probability of the system to transit to state s' from s by the action a. This p function $\forall s, s'$ is represented as P_a function which is a matrix of size $|S| \times |s| \times |A|$. Here the sum of all probabilities $\Sigma_s p(s'|s, a) = 1$. The $p()$ probability distributions over the next state s' follow the fundamental property which describes them as a MDP. If h_t represents the history of states and actions until time step t, for a MDP it is true that

$$\forall h_t, a_t, s_{t+1}, P(s_{t+1}|h_t, a_t) = P(s_{t+1}|a_{t+1}) = p(s_{t+1}|s, a) \tag{3}$$

As a result of the action a in state s at time y, the agent receives a reward $r_t = r(s, a)$. Here, we consider positive value of r_t as gains and negative value as costs. For our study, we assume that the state transition and reward functions are $stationary$, hence they do not change from one time step to another.

To solve the MDP, the task is to solve for a policy that optimizes the rewards. Although there are many criteria in literature[3] that evaluate the policy, we will use the $(\gamma-)$discounted criterion which is one of the most commonly used infinite horizon criterion. The finite, discounted, total reward, and average criteria defines a *value function* For a given policy π. The value function for the discounted criteria, for a given policy *pi* assuming $0 \leq \gamma < 1$ is given by

$$V_\gamma{}^\pi(s) = E^\pi[r_0 + \gamma r_1 + \gamma^2 r_2 + \gamma^3 r_3 + ... + \gamma^t r_t + ..|s_0] \tag{4}$$

With this method, it is possible to establish the *Bellman optimality principle*[4]. This allows us to use the dynamic programming approach to solve MDPs efficiently. The goal of the MDP solver is to finding π^* such that

$$\pi^* \in argmax V^\pi \tag{5}$$

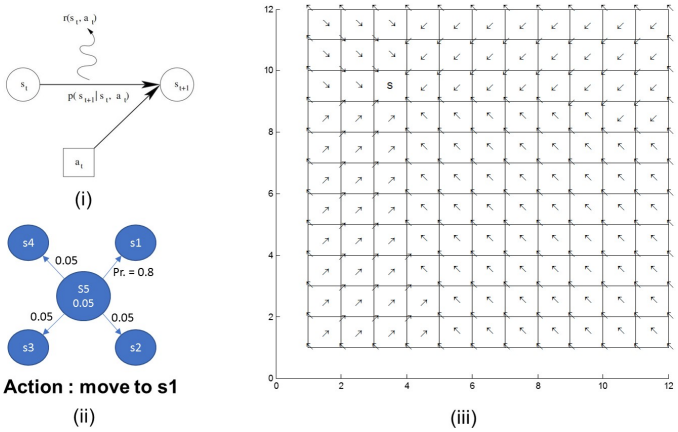


Fig. 3 (i)Markov Decision Process [3] (ii) An example of an action to be performed and transition probabilities. s1,s2,s3,s4 denotes the NE, SE, SW, and NW directions and s5 denotes the current state (iii) Solution to the MDP problem denoting actions at every state that can lead the robot to the goal s

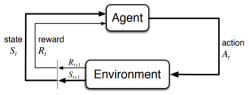


Fig. 4 Reinforcement Learning framework ([6])

To solve this MDP, we use the $MDPtoolbox$ toolbox for MATLAB[5]. A value iteration method is used to solve the Bellman's equation iteratively.

$$\pi(s) := argmax_a\{\sum_{s'} P_{\pi(s)}(s,s')(R_{\pi(s)}(s,s') + \gamma V(s'))\} \tag{6}$$

$$V(s) = \sum_{s'} P_{\pi(s)}(s,s')(R_{\pi(s)}(s,s') + \gamma V(s')) \tag{7}$$

where P is the transition matrix and R is the reward matrix.

For validation, a deterministic case with fixed transition probabilities were used to generate actions for each state in the system to reach a goal. The simulation can be seen in Fig. 3

IV. Reinforcement Learning

As showed in the previous section, applying the MDP path planning solution directly on the actual system works well when the system is following nominal dynamics. Still, there is some stochasticity in the transition dynamics due to the mismatch in the position estimated by the camera, and the actual position of the robot. This mismatch can still be taken care of by the finite state-action MDP model by incorporating the deviations due to position mismatch in the transition probability of the MDP. That is, we can still assume that the system has finite states (grid points) and finite actions (NE,SE,SW,NW), such that the knowledge of the stochasticity due to position mismatch can be represented as stochastic transitions $< s, a, s' >$.

Reinforcement learning (RL) is inspired by the way animals learn through experience how to behave in an environment. Fig(4) shows basic idea behind RL. The agent chooses an action, and records the reward it gets, and the state it reaches. Based on the results it gets, it tries to change its action behavior to get more reward. Hence, RL algorithm's objective is to learn the optimal policy which the agent should follow to maximize its (discounted) cumulative reward. Main elements of RL schemes are as follows ([6], [7]): 1) **policy**: determines what action is chosen

by the agent at a state. It can be represented as a function or a lookup table, and can be deterministic or stochastic in nature. The objective of the RL algorithm is to eventually find the optimal policy. 2) **reward function** is based on the goal of the agent. Its purpose is similar to what is known as the 'running cost' in the cost function of optimal control problems. It maps every transition to some scalar value that acts as a reward to the agent. The agent thus attempts to maximize its cumulative reward by finding out the best policy. 3) **value function**: predicts the future reward. There are two types: the state value function V(s), and the action value function Q(s,a), where s represents the state and a represents action. State value function is used when the model of the environment is known, whereas action value function is preferred when the model of the environment is unknown. Note that, a reward function evaluates the immediate reward, while the value function is the prediction of the total reward.

There are three main methods to solve RL problems: ([6], [7]) 1) **Dynamic programming (DP)**: is mathematically well developed, but needs the actual model of the environment. 2) **Monte Carlo (MC)** methods: are completely model free, but they are difficult to implement online as these methods are not incremental, and are episodic instead. 3) **Temporal difference (TD)** methods: have the best of both DP and MC methods. They are model free unlike DP and are easy to implement online with step by step computation. From the above description, we can say that choosing TD methods for our problem would be the best decision as our problem involves both lesser knowledge of the environment model and online implementation.

A. Action value function (Q-function)

Action value function can be written as:

$$Q_\pi(s,a) = E_\pi[\sum_{k=0}^{\inf} \gamma^k r_(t+k+1)|S_t = s, A_t = a] \tag{8}$$

where s represents the state and a represents action, $\gamma \in (0,1]$ is the discount factor, r_t is the reward at time t, π is the policy according to which a is chosen at every time step t. For the optimal policy π_*, we get the maximum action value as:

$$Q_*(s,a) = \max_p iQ_\pi(s,a) \tag{9}$$

which can be expressed in terms of the state value function as:

$$Q_*(s,a) = E_\pi[r_(t+1) + \gamma V_*(S_(t+1))|S_t = s, A_t = a] \tag{10}$$

Since,

$$V_*(s) = \max_a Q_(\pi_*)(s,a) \tag{11}$$

$$Q_*(s,a) = E_\pi[r_{t+1} + \gamma \max_a{}' Q_*(S_{t+1}, a')|S_t = s, A_t = a] \tag{12}$$

This gives us the action value iteration equation, using which RL algorithms like Q-learning and SARSA are derived by employing TD methods.

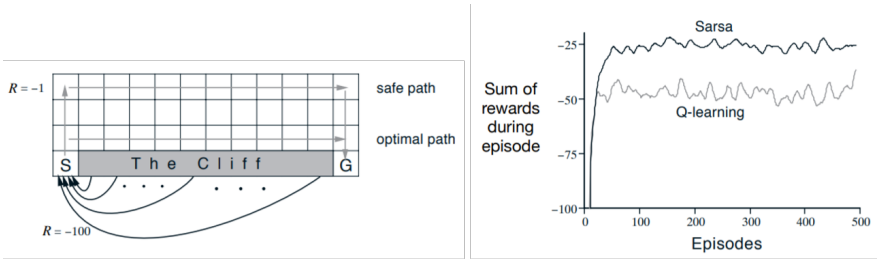
B. Q-learning v/s SARSA

Both Q-learning and SARSA are based on TD methods. The main difference between these algorithms is that SARSA is **on-policy**, whereas Q-learning is **off-policy**, i.e., SARSA estimates the return for state-action pairs assuming the current policy continues to be followed, whereas Q-learning estimates the return (total discounted future reward) for state-action pairs assuming a greedy policy were followed instead of the actual policy being followed. Mathematically, the SARSA update equation looks like:

$$Q(s_t,a_t) \leftarrow Q(s_t,a_t) + \alpha[r_{t+1} + \gamma Q(s_{t+1},a_{t+1}) - Q(s_t,a_t)] \tag{13}$$

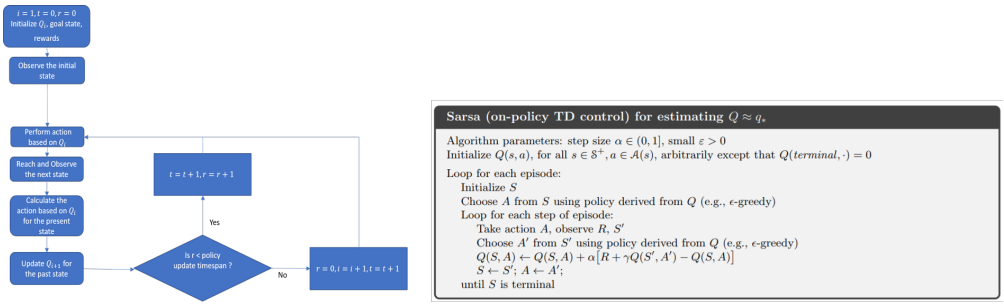
Whereas Q-learning update equation is:

$$Q(s_t,a_t) \leftarrow Q(s_t,a_t) + \alpha[r_{t+1} + \gamma \max_a Q(s_{t+1},a) - Q(s_t,a_t)] \tag{14}$$



(i) (ii)

Fig. 5 (i) Cliff Walking problem [6] (ii) Total Rewards v/s episodes, Cliff walking problem [6]



(i) (ii)

Fig. 6 (i) Flowchart of our algorithm, (ii) SARSA [6]

Thus, the mathematical difference comes in terms of the action value function for s_{t+1} corresponding to the current policy's action a_{t+1}, or the greedy policy's action $argmax_a Q(s_{t+1}, a)$ irrespective of what the current policy is.

Consider the cliff-walking grid world problem given in [6] (see Fig(5)(i)): 1) in case of Q-learning, the agent eventually learns values for the optimal policy, making it travel along the edge of the cliff, but occasionally falls off the cliff because of $\epsilon - greedy$ action selection. 2) whereas, in case of SARSA, the agent takes action-selection into account, learns the longer but safe path, consequently collecting more rewards in the intermediate episodes than Q-learning (see Fig(5)(ii)), making it a better choice for online implementation. Still, both the algorithms eventually converge at the same solution. Thus, SARSA seems to be a better choice for solving our problem as it is based on real-time hardware implementation.

C. SARSA

We are going to use the algorithm shown in Fig(6)(ii) for our RL agent, with a little modification. Flowchart of our version of the algorithm is shown in Fig(6)(i). The difference is in the fact that the algorithm shown in Fig(6)(i) uses the updated Q (updated at each t) to update the policy being used at each t, whereas our algorithm also updates Q at each t following the same procedure, but the policy being used is updated after some fixed time window (policy update timespan). This provides us with more freedom to change the way the algorithm works. Another motivation is that updating the policy at every time step may lead to biased Q estimates, whereas learning the Q estimates by fixing the policy for some time window can average out the biasness and give nice results. Again, finding out the most suitable time window is empirical in nature.

Note that: $\alpha \in (0, 1]$ is the learning rate, that decides how much previously learned information is retained. $\gamma \in (0, 1]$ is the discount factor, which sets the preference for immediate reward or more future-oriented reward. From [6], we know that "SARSA converges with probability 1 to an optimal policy and action-value function as long as all state–action pairs are visited an infinite number of times and the policy converges in the limit to the greedy policy." (see Fig(7)(i)) i.e., as $\epsilon \to 0$, for example: $\epsilon = 1/t$.

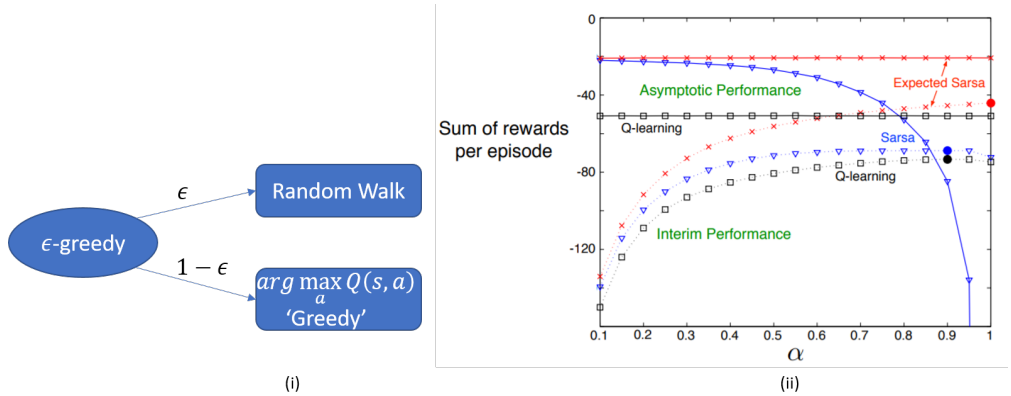


Fig. 7 (i) ϵ-greedy policy: chooses random walk with probability ϵ (ii) Performance of various TD learning algorithms [6]

For the cliff walking problem [6] (see Fig(7)(ii)) SARSA can only perform well in the long run at a small value of α, at which short-term performance is poor. Accordingly, for our problem,

$$\epsilon = 1/e^{(3*episode\#)/(max.episode\#limit)} \tag{15}$$

$$\alpha = 1/e^{((4*episode\#)/(maxepisode\#limit))} \tag{16}$$

D. Environment and RL Algorithm parameters

We are considering two types of environment models for our simulations: deterministic MDP, and Stochastic MDP. Information about the transition probabilities is shown in Fig(12). Other than that, for both the types: reward at the boundaries = -500, reward at the goal state (1,1) = 500. For deterministic: reward per transition = -5. RL algorithm parameters: total no. of episodes = 500, policy update window = 5, no. of Policy-update limit = 100, $\gamma = 0.9$.

E. Cases and Results

Clearly, the results show that the Q estimate that the agent is learning, seems to be converging towards the actual Q function. Fig(14) is showing the results for 11x11 coils deterministic case 1: the first plot shows the intermediate episode result, the second one shows the final result, and the third one shows the convergence of total reward per episode with respect to no. of episodes. The same order follows for Fig(15), Fig(16), Fig(17), and Fig(18), which are the results for 11x11 coils deterministic case 2, 11x11 coils stochastic case, 4x4 coils deterministic case, and 4x4 coils stochastic case, respectively. The probabilistic transition rewards for the 11x11 stochastic case and the 4x4 stochastic case are shown in Fig(13).

Key take-aways: The 11x11 deterministic case 1 takes a total of 28143 time steps for completing 500 episodes. Total time steps for 11x11 deterministic case 2: 26875, for 11x11 stochastic case: 60463, for 4x4 deterministic case: 5149, for 4x4 stochastic case: 8544. For the hardware setup, the robot takes around a second for a single transition. Therefore, implementing the 11x11 study cases on the hardware seems inconvenient, as the purpose of these experiments is just to show a proof of concept. Thus, choosing 4x4 study cases for hardware implementation seems the right decision.

The 11x11 deterministic case 1's results look promising, but still does not correspond to the optimal one. To improve, one can certainly increase the no. of episodes as much as they want, and consequently reach the true Q values. But, increasing the episodes means increased simulation time, which can be impractical.

One reason for 11×11 deterministic case 1's sub-optimal solutions can be that: 1) the negative reward per transition is pretty low in magnitude compared to the high positive goal reward, 2) a high γ , resulting in the consideration of more priority to future (delayed) reward. Due to these reasons, the algorithm is estimating the Q values in such a way that the agent is less cautious about incurring the small negative rewards in the way towards getting a delayed large positive reward. In order to check our above given hypothesis, we simulated the 11x11 deterministic case 2 (reward per transition

= -15) with higher negative reward per transition as compared to 11x11 deterministic case 1 (reward per transition = -5). As expected, we can see that 11x11 deterministic case 2 has done much better than 11x11 deterministic case 1, so much that case 2's solution is the exact optimal solution at least for the initial condition shown in their respective plots.

V. End-to-End Learning: From pixels to action

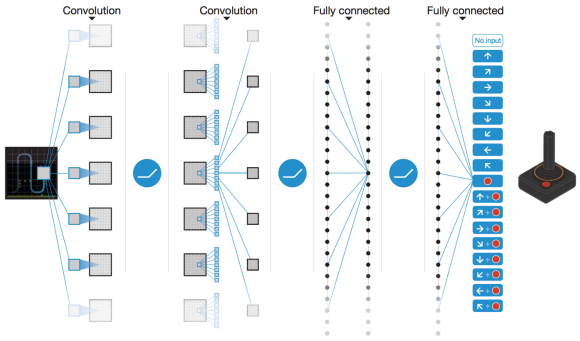


Fig. 8 a. DRL approach for control of micro-robots

A. Deep Reinforcement Learning

With recent advances in Deep learning, there is an explosion of applications where vision based control is possible than before. The first paper on using Deep learning for Reinforcement learning called the approach as Deep Reinforcement learning for using a Q-network for approximating the state-action pair values. In essence, deep learning is a machine learning algorithm for learning representations. It is used for low dimensional approximation of infinite dimensional spaces. This opens up a new area of Artificial Intelligence where agents can act based on rewards from environment with a perception input alone. People in deep learning/ AI community call this as an end-to-end learning problem where an agent learns a complicated policy from simple interactions with the environment and without explicitly stating the relations. The absence of hand-crafted features make this kind of learning approach unique from others.

However, the problem format of reinforcement learning or sequential decision making problem posts several challenges to implement deep learning successfully. In real RL problems, the rewards are delayed or sometimes only available at the end of the event. This sparse feedback is not good enough to train a deep learning network.

B. A note on Approximate dynamic learning

When the size of the problem is sometimes very large, complex and stochastic, the usual method of dynamic programming suffers from the curse of dimensionality. This is well known even at the time of development by Bellman. The theory of approximate dynamic programming tries to address this problem through the use of different functional approximation techniques like approximating value function or Q-function, monte carlo approximation of expections, working on subset of actions etc. The curse of dimensionality we face in this problem is sometimes referred to as *factored representation*. If the state variables are discrete multidimensional vector of discrete elements, it is not practical to list all the possible combinations of state variable in a flat representation. This is exactly the problem we have in hand - to map an image input to a fixed point as defined by Bellman operator which is the value of the state or value of the state-action pair. Considering a RGB image from a camera which has $300 \times 300 \times 3$ pixels with each pixel having $0 - 255$ values, there are 256^{270000} states that are to be considered. A standard Q learning algorithm is known to converge only after each state is visited infinite number of times. It is thus practically impossible with all supercomputers together to solve this problem. Thus, we try to make an approximation to a lower dimensional space where it is still solvable. Deep learning techniques offer support in terms of providing this approximation.

In a typical intractable problem setting, we start with a parametric function approximator for value function and apply dynamic programming. Considering that the estimated value function is a good approximation, we train the

parameters of the approximation using the Bellman optimality equation. This step is called training or back propagating the error to the parametric function. The process is continued till the error diminishes. An obvious question in mind at this point would be on the proof of convergence of such function approximator based value function to the optimal value function. The proof follows from the Contraction mapping theorem:

*Theorem:*For any metric space M that is complete (ie. closed) under an operator T(v), where T is a γ-contraction, then T converges to a unique fixed point at a linear convergence rate of γ.

Proof: Define the Bellman expectation backup operator T^π, $T^\pi(v) = R^\pi + \gamma P^\pi v$. This operator is a γ-contraction, ie, it makes the value functions closer by at least γ, which is given by:

$$\begin{aligned} ||T^\pi(u) - T^\pi(v)||_\infty &= ||(R^\pi + \gamma P^\pi u) - (R^\pi + \gamma P^\pi v)||_\infty \\ &= ||\gamma P^\pi (u - v)||_\infty \\ &\leq ||\gamma P^\pi ||(u - v)||_\infty||_\infty \\ &\leq \gamma ||(u - v)||_\infty \end{aligned}$$

Note, we measure the distance between state-value function u and v by the ∞ norm as the largest difference between the state values, $||u - v||_\infty = \max_{s \in S} |u(s) - v(s)|$ This results described in this section agrees to the theory of General Policy Iteration (GPI) which applies to any reinforcement learning algorithm. In simple words, GPI proves that one can use any policy evaluation algorithm to estimate value function and do a policy improvement based on the value function with an asymptotic convergence to the optimal policy and value function. One can not achieve an optimal policy without achieving a convergence on the value function iteration.

C. Simulator environment

Following the problem statement, we start with building the simulated environment using Python. Here, the environment is designed in a parameterized approach to give greater flexibility in future. To comply with the experimental setup, all experiments are done in a 11*11 coil environment. In all, this corresponds to 23*23 states. Since the original problem is a Partially Observed Markov Decision Process (POMDP), we have two kinds abstraction. The underlying simulator works as per the hidden markov states where as the agent is able to see only the observation which is nothing but the coils and the robot itself. In order to be able for the agent to reach the goal, we give an additional signal of color indicating the goal position. It is important to note that the robot is not explicitly given any sort of information other than the observed image which is almost similar to the camera image. We drew this inspiration to build a simulated environment from examples of openai gym.

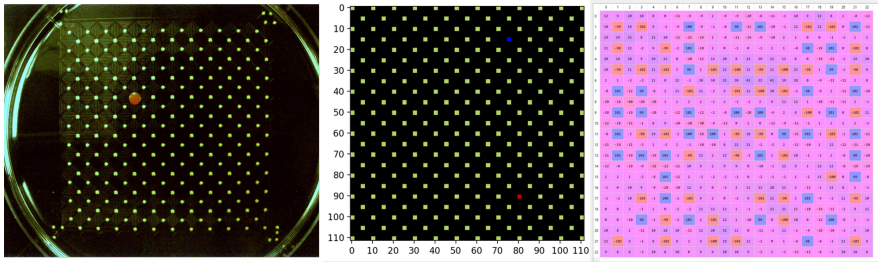


Fig. 9 (i)Actual image from camera showing robot. (ii) Simulated image output - Observation state. (iii) Hidden state of coils showing magnetic potentials

D. Deep Q Network (DQN) algorithm

Taking inspiration from [1], in this work, we tried to implement a Deep Q learning algorithm for control of micro-robots as described in the problem statement. The deep reinforcement learning approach here takes an image of

size 112*112 and outputs action probabilities for 9 discrete actions.The heart of the algorithms is a Q-network which is a Convolution Neural Network that can process the images. The implementation follows the same exact configuration as the one in literature [1]so as to start with a tested procedure. Due to the simplicity among all DRL algorithms, we chose to experiment starting with DQN. It was later understood that DQN suffers greatly from initial learning, over-fitting and low rate of convergence. Given the available time, we still feel the justification of successful implementation of the algorithm.

Algorithm 1 Deep Q-learning with Experience Replay

Initialize replay memory $\mathcal{D}$ to capacity N
Initialize action-value function Q with random weights
for episode $= 1, M$ **do**
  Initialise sequence $s_1 = \{x_1\}$ and preprocessed sequenced $\phi_1 = \phi(s_1)$
  for $t = 1, T$ **do**
    With probability ϵ select a random action a_t
    otherwise select $a_t = \max_a Q^*(\phi(s_t), a; \theta)$
    Execute action a_t in emulator and observe reward r_t and image x_{t+1}
    Set $s_{t+1} = s_t, a_t, x_{t+1}$ and preprocess $\phi_{t+1} = \phi(s_{t+1})$
    Store transition $(\phi_t, a_t, r_t, \phi_{t+1})$ in $\mathcal{D}$
    Sample random minibatch of transitions $(\phi_j, a_j, r_j, \phi_{j+1})$ from $\mathcal{D}$
    Set $y_j = \begin{cases} r_j & \text{for terminal } \phi_{j+1} \\ r_j + \gamma \max_{a'} Q(\phi_{j+1}, a'; \theta) & \text{for non-terminal } \phi_{j+1} \end{cases}$
    Perform a gradient descent step on $(y_j - Q(\phi_j, a_j; \theta))^2$
  end for
end for

Fig. 10 Deep Q-Learning model free algorithm. [1]

E. Implementation details of Deep Q learning

Since no standard toolbox was available to use to call the algorithm, we resolved to do an implementation using Python. We also used Tensorflow deep learning framework as a backend. TensorFlow offers in build APIs for writing deep learning applications. Our implementation of the Q network had four layers : the first convolutional layer with 16 convolution filters of 8*8 size and stride 4 followed by rectifier nonlinearity. The second hidden layer convolves 32 4*4 filters with stride 2 followed by rectifier non-linearity. The final hidden layer is fully connected with 256 relu units. The output layer is a fully connected linear layer with single output for each action. In this study, we considered nine action outputs - each one step movement in directions NW-N-NE-E-SE-S-SW-E and one for being stationary. The cost function used was to minimize the squared error between the actual q-value and the expected q-value from the functional approximation. We use a RMSPropOptimizer as the optimization algorithm to train the deep learning model. The forward pass gives out the probabilities for choosing each action and the highest probability is chosen for execution.

F. Results and discussion

To test the working of the implementation, we did run a simulation of 3000 episodes. The adjoining figure shows the distribution of rewards across different episodes. Since there is no visible trend seen, we can not conclude on the improvement over time. We do believe the implementation is flawless in the sense there were no other interruptions in the prescribed amount of runs. We also verified if the training is happening without diverging. One of the insights from this experiment is that there are very less number of good examples to learn and most of the learning currently is from failed examples.As per literature, the number of iterations should reach around 10^7 for a sub-optimal performance. We can not exactly prove convergence to optimal value in this kind of approximation setting. The simulator build in first step proves good training ground for doing RL research for microrobot problems. In future, we are planning to extend the control to many independent robots so that each can make their own control. It would interesting to see how they either independently/ cooperatively achieve tasks. Further, we also noticed that the idea of giving terminal state rewards in this problem is quite different from one implemented in literature where the agent collects periodic rewards.The shaping of reward function will be an immediate focus.

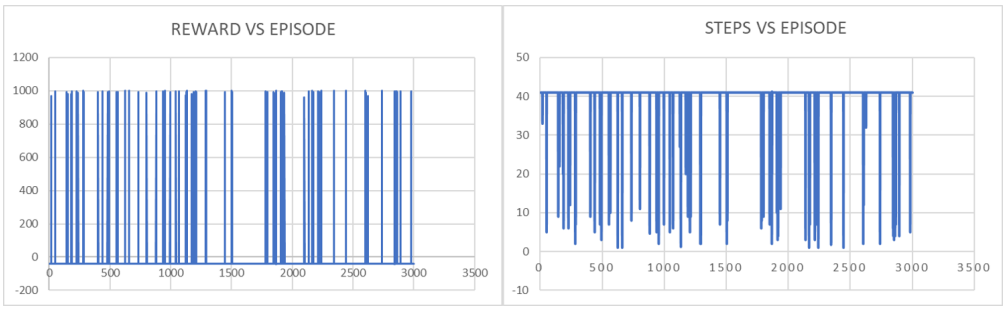


Fig. 11 Figure showing rewards and steps. Rewards higher than zero represents the attainment of goal position. Steps less than 50 represents the goal achievement. Total episodes = 3000

VI. Contribution

As a team, we had regular meetings on Friday to discuss on literature review, problem suggestion and formulation. We had lab sessions for learning on the hardware implementation. Towards the end of project, we had individual goals to succeed in this project viz. hardware implementation by Benjamin, SARSA implementation by Shubhankar and Deep Q learning algorithm implementation by Kumaraguru.

VII. Conclusion

In this work, we explored applying dynamic programming, reinforcement learning, approximate dynamic programming in the context controlling micro–robots. We had thrilling moments during the phase of the project - to see successful running of DQN algorithm, to get a convergence on SARSA, to see actual robot in action. The foundations we learned from AAE-568 course was helpful in extending our knowledge and understanding to apply in a new context for a real world research problem.

Acknowledgments

We would like to thank Prof. David J. Cappelleri and the Multi-scale robotics and Automation Laboratory in the School of Mechanical Engineering at Purdue University for the experimental setup.

References

[1] Mnih, V., Kavukcuoglu, K., Silver, D., Graves, A., Antonoglou, I., Wierstra, D., and Riedmiller, M. A., "Playing Atari with Deep Reinforcement Learning," *CoRR*, Vol. abs/1312.5602, 2013. URL `http://arxiv.org/abs/1312.5602`.

[2] Chowdhury, S., Jing, W., and Cappelleri, D. J., "Towards independent control of multiple magnetic mobile microrobots," *Micromachines*, Vol. 7, No. 1, 2015, p. 3.

[3] Sigaud, O., and Buffet, O., *Markov decision processes in artificial intelligence*, John Wiley & Sons, 2013.

[4] Bellman, R., "The theory of dynamic programming," *Bulletin of the American Mathematical Society*, Vol. 60, No. 6, 1954, pp. 503–515.

[5] Chadès, I., Chapron, G., Cros, M.-J., Garcia, F., and Sabbadin, R., "MDPtoolbox: a multi-platform toolbox to solve stochastic dynamic programming problems," *Ecography*, Vol. 37, No. 9, 2014, pp. 916–920.

[6] Sutton, R. S., and Barto, A. G., *Reinforcement learning: An introduction*, Vol. 1, MIT press Cambridge, 1998.

[7] Khan, S. G., Herrmann, G., Lewis, F. L., Pipe, T., and Melhuish, C., "Reinforcement learning and optimal adaptive control: An overview and implementation examples," *Annual Reviews in Control*, Vol. 36, No. 1, 2012, pp. 42–59.

Appendix

A. Figures

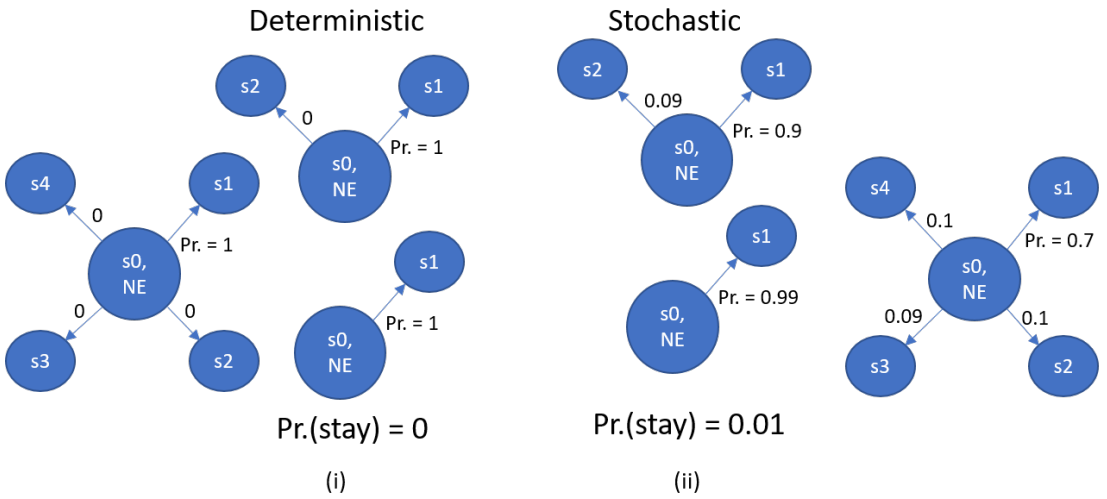


Fig. 12 (i) Deterministic type (ii) Stochastic type environment

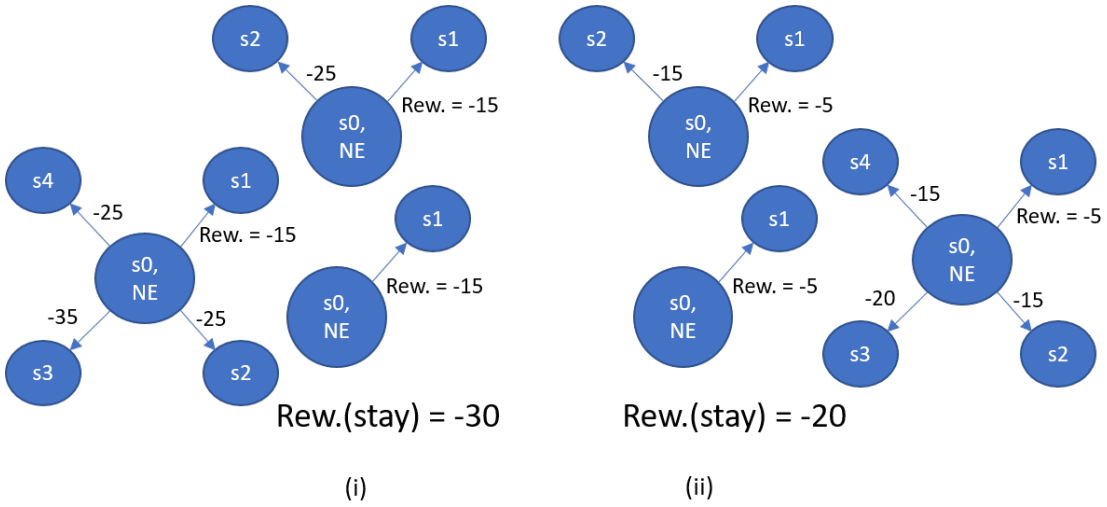


Fig. 13 (i) 11x11 stochastic case transition rewards (ii) 4x4 stochastic case transition rewards

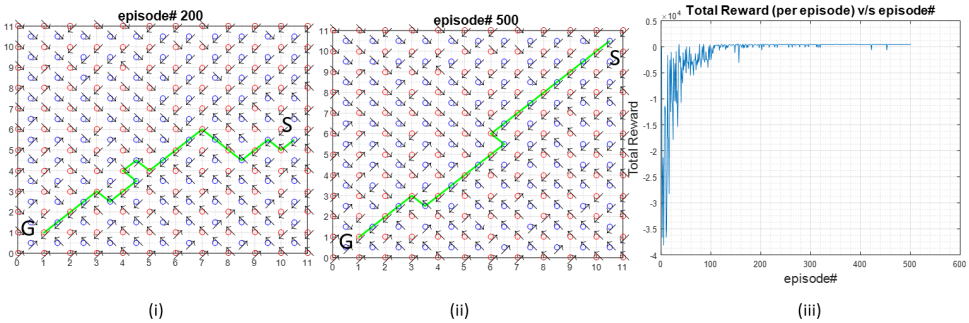


Fig. 14 11x11 deterministic case 1 (i) at episode 200 (ii) at the last episode (iii) Total reward v/s episode no.

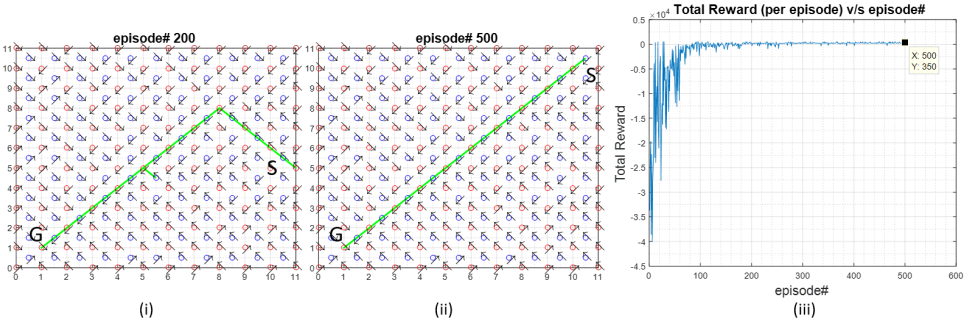


Fig. 15 11x11 deterministic case 2 (i) at episode 200 (ii) at the last episode (iii) Total reward v/s episode no.

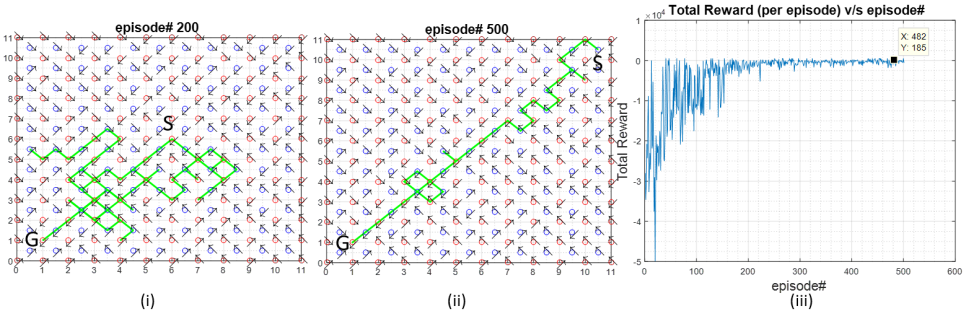


Fig. 16 11x11 stochastic case: (i) at episode 200 (ii) at the last episode (iii) Total reward v/s episode no.

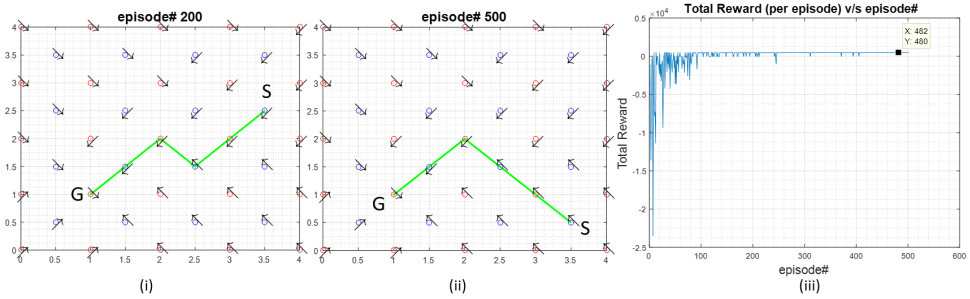


Fig. 17 4x4 deterministic case: (i) at episode 200 (ii) at the last episode (iii) Total reward v/s episode no.

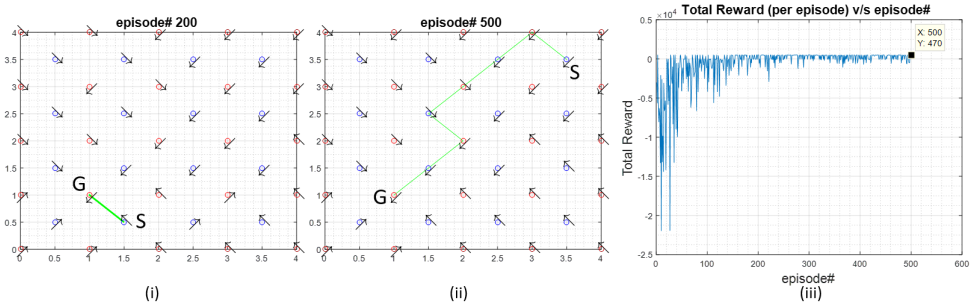


Fig. 18 4x4 stochastic case: (i) at episode 200 (ii) at the last episode (iii) Total reward v/s episode no.