

Simultaneous Multiagent learning in an adversarial setting



B.TECH PROJECT REPORT (EE492), Spring 2010

Submitted on April 16, 2010 by

Rohit Vaish (Y6402), Department of Electrical Engineering, IIT Kanpur

&

Sandip Kumar Gupta (Y6425), Department of Computer Science & Engineering, IIT Kanpur

Under the supervision of :

Dr. Amitabha Mukerjee, Department of Computer Science & Engineering, IIT Kanpur

&

Dr. K.S.Venkatesh, Department of Electrical Engineering, IIT Kanpur

ACKNOWLEDGEMENTS

The authors would like to thank their project supervisors - Dr. Amitabha Mukerjee and Dr. K. S. Venkatesh for providing us the opportunity to work on this task and their support and guidance throughout the course of this project, without which this study would not have been possible.

List of Figures

1	Keepout setting	3
2	Tile Coding (Image source [1])	11
3	Loop depicting flow of information in the system	12
4	Simulation Window snapshot	14
5	Action space for the given problem. White lines depicting agents' distance from the target and direction of heading are shown. Five of the seven action points have been shown on the broken red line passing through target point. The robots shown belong to CMDragons'02 Robot Soccer team (Image source [2])	15
6	Comparing Value estimation learning rates α	17
7	Comparing attacker and defender speeds	18
8	Comparing Angular separation and agent velocities for (a) rewards = R_0 and (b) our reward formulation	19
9	Class Diagram for 2 agent GRAWoLF Implementation	23
10	Finding out robot position and velocity with the help of overhead tag. Given view is from the overhead camera.	24

Acronyms

GRAWoLF : **G**radient Based **W**in or **L**earn **F**ast

RL : **R**einforcement **L**earning

SARSA : **S**tate-**a**ction-**r**eward-**s**tate-**a**ction

CMACS : **C**omputational **M**odeling and **A**nalysis for **C**omplex **S**ystems

SDL : **S**imple **D**irectmedia **L**ayer

MDP(s) : **M**arkov **D**ecision **P**rocess(es)

Contents

1	Introduction: Learning in stochastic games	2
2	Objective	3
2.1	The Setting	4
3	Literature Survey	5
4	Basic Theory	6
4.1	Markov Decision Processes	6
4.2	Matrix Games	6
4.3	Multiagent stochastic Games	7
4.4	GraWoLF	7
4.4.1	Policy Gradient	8
4.4.2	Variable Learning Rate (δ_k) Selection: Win or Learn Fast	9
4.4.3	Important Observations	10
4.5	Tile Coding	11
5	Methodology Adopted	12
5.1	System setup	12
5.2	Proposed Reward Function Formulation	13
5.2.1	Rewards/Penalties for Attacker	13
5.2.2	Rewards/Penalties for Defender	13
6	Robot Simulation	14
6.1	Implementation Details	14
7	Results	16
7.1	Learning v/s Random Trials	16
7.2	Comparing Value estimation learning rates α	17
7.3	Comparing Attacker and Defender Speeds	18
7.4	Comparing Angular separation and agent velocities	19
8	Conclusions	20
9	Scope for future work	21

Abstract

The aim of this study is to explore the phenomenon of *adversarial learning* in multiple agent stochastic game of *Keepout*. We intend to investigate the effectiveness of reinforcement learning(RL) algorithms in stochastic games, with multiple agents learning in an adversarial setting, as against those who choose their policy randomly. As an example, we work on the GraWoLF (Policy gradient based win or learn fast) algorithm, and try to demonstrate the improvements attained with RL through simulations. We also propose a reward function formulation and gauge its usefulness in the current learning scenario. A collection of results over varying values of parameters such as learning rate(s), tiling configuration, rewards etc. has been presented. This report presents the work done in this regard upto now and associated future plans.

1 Introduction: Learning in stochastic games

Multi-robot learning refers to problem of learning to act or behave in a setting consisting of multiple robots. Now, the other robots, who have their own set of goals and tasks to achieve, might be learning as well. The actions of one particular agent, therefore, are no more stationary but depend on the action-space of the other agents in the environment, violating the Markovian assumption that significantly simplifies single agent learning task. The problem is further complicated by presence of continuous state and action spaces and minimal training data. Multi-robot learning, hence, combines all these problems into the problems of learning in robots in a continuously fluctuating environment. Moreover, in a society of agents (multiple agent setting), each individual agent need not learn every task by virtue of its own discovery. Instead, many agents work together to cooperate and accomplish a given task. In this process, they exchange information & knowledge with each-other and learn from their fellow agents.

Through this study, we seek to explore the phenomenon of adversarial learning in a multi-agent setting consisting of multiple robots participating in a game of *Keepout*. Stochastic games are a natural extension of *Markov Decision Processes* (MDP) to multiple agents and have been well-researched through a Game Theoretic perspective. A traditional solution that has emerged towards for finding optimal policies for multiple agents at the same time is that of **Nash Equilibrium**. Here each player is assumed to know the equilibrium strategies of the other players, and no player has anything to gain by changing only its own strategy unilaterally.

An important question could be as to how reinforcement learning agents be *cooperative*. Tan [3] identifies three ways of answering this, viz:

- Agents can communicate instantaneous information such as sensation, actions or rewards.
- They can communicate episodes that are sequences of (sensation, action, reward) triples experienced by them.
- Agents can communicate learned decision policies.

We try to find out the effect on agent behavior under the proposed reward formulation in the setting explained in [2]. Conclusions from such a study could supplement our understanding of multi-agent learning, besides providing significant insights in conducting related studies in future.

2 Objective

We aim a two-fold approach to conduct this study. One, by simulating a multi-agent setting in a virtual environment to judge the effectiveness of our strategies and learning algorithms. Second, by performing these experiments on a team of real robots, to depict the effects of cooperation.

The game of Keepout (Fig. 1) consists of two teams - Attackers & Defenders - with at least one member per team present on the arena. Any one of the attackers have to reach a ball kept inside a target zone on the arena, and the defenders have to prevent the attackers from doing so, for as long as possible. In this process, the defender can block/impede any attacker's path, but not essentially *push* it in doing so. Similarly, the attacker may not push the defender. Each such run is limited by a time frame T_0 , inside which each robot has to accomplish its task. Hence, each run lasts for atmost T_0 time, after which the run is aborted (in case the attacker is not able to reach the ball in that time), and a new run commences. For the purpose of simulation, it shall be assumed that any collision between attacker and defender while the trial is still running will mean defender has been able to successfully 'impede' the attacker and the run stops there.

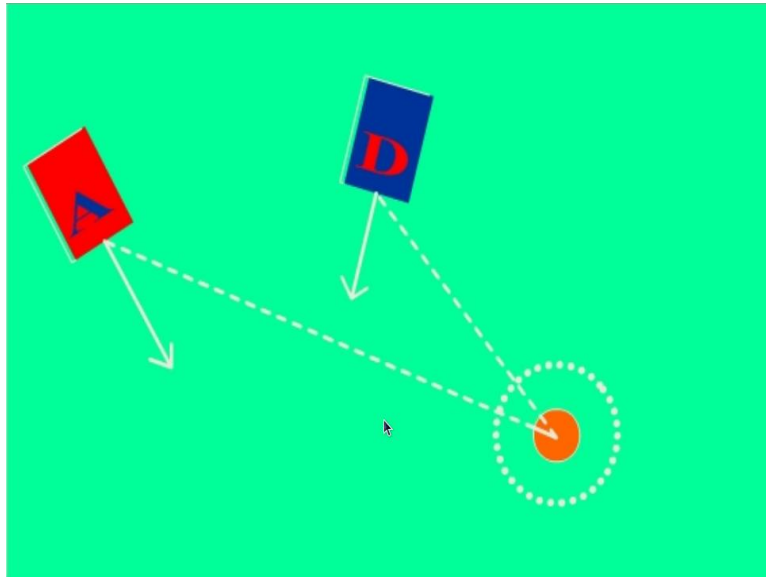


Figure 1: Keepout setting

2.1 The Setting

The setup for the aforementioned problem consists of an overhead camera mounted over the field, which serves the purpose of global vision system. The attacker is normally assigned a random initial position, but care is taken so that the defender is closer to the ball, to start with. Both robots are wirelessly connected and controlled by separate computer systems, without any human intervention, thereby making them completely autonomous.

With the given setting, we shall try to explore/answer the following pertinent questions:

- How well do constantly learning agents fare against their fixed-strategy counterparts?
- In a game of learning agents, how well does a team with cooperative strategies fare against a team of independent agents?
- Exploring various learning mechanisms under the same cooperative setting, and judging their relative goodness.

3 Literature Survey

Multi-agent stochastic systems have been studied in great detail in the past - both from a Game Theoretic point of view & in the reinforcement learning domain. Bowling and Veloso [4] provide a comprehensive presentation of relevant techniques for both of these areas. Lauer & Reidmiller [5] present a distributed variant of Q-learning that allows to learn the optimal cost-to-go function in stochastic cooperative multiagent domains *without* communication between the agents.

Commensurately, several multi-agent systems have been developed for speed and accuracy. The MALE system (Sian 1991) uses an interaction board (similar to a blackboard) to coordinate different learning agents. DLS (Shaw & Sikora 1990) use distributed problem-solving approach to rule induction by dividing data among inductive learning agents. Whitehead (1991), in his theorem, states that n reinforcement learning agents who can observe everything about each other can decrease the required learning time at a rate that is $\Omega(1/n)$.

Similarly, related work in Distributed Artificial Intelligence has addressed the issues of organization, coordination and cooperation among agents, but not for multi-agent learning. Extensive work has also been done in studying general-sum games (like Predator-Prey problem) for cooperation [3]. Issues such as communication language, agent-beliefs, resource constraint and negotiation have not been addressed in this study. Besides, all agents are essentially assumed homogenous.

4 Basic Theory

Multiagent stochastic games are at the fusion of two separate fields of study, namely Markov Decision Processes and Matrix games.

4.1 Markov Decision Processes

Markov Decision Processes (MDPs) are useful in dealing with single agent learning in an environment. The environment is static otherwise, and only the agent brings about a change in it. The agent on the basis of its current state, takes an action to change the environment so as to maximize its reward.

Formally Markov Decision Processes can be described as a tuple (S,A,T,R) , where:

- S is the set of states
- A is the set of actions available to the agent
- T is a transition function, mapping $S \times A \times S \rightarrow [0, 1]$. It describes the mapping from the agent initially being in a given state, choosing a particular action and then attaining some other final state, to the probability space. (state-action-state space to probability)
- R is the reward function which maps the action of the agent, given a state, to a reward. Hence the mapping is $S \times A \rightarrow \mathbb{R}$

4.2 Matrix Games

Matrix Games deal with multiple agents involved in decision making and the reward of any agent depends on the joint action of all other agents. It can be formally represented as a tuple $(n, A_{1\dots n}, R_{1\dots n})$, where

- n is the number of agents involved
- A_i is the possible action of i^{th} agent
- R_i is the reward of i^{th} agent, depending on the actions of all agents.

Hence its a mapping from $A_1 \times \dots \times A_n \rightarrow \mathbb{R}$

The matrix games can be zero sum or general-sum. In zero sum matrix games, the sum of rewards of all agents amounts to zero. The optimal strategy for any agent is found using the concept of *Nash Equilibrium*.

Perhaps the most striking feature of, and the reason why most multiagent problem are formulated on the zero sum games, is that they have the advantage of a **unique nash equilibrium**.

4.3 Multiagent stochastic Games

Multiagent stochastic games is a fusion of Markov Decision Processes and Matrix games in the sense that it is a multiagent setting in an environment with given states. The reward here is dependent on the current state of the *environment*, the joint action space of all the agents and the next state of the environment (Hence a fusion of MDP and Matrix games).

Formally, it can be described as tuple $(n, S, A_{1\dots n}, T, R_{1\dots n})$, where

- n is the number of agents
- S is the set of states of the environment
- A_i is the set of actions of i^{th} agent
- T is the transition function mapping the probabilities of an action in given state, and can be written as $S \times A \times S \rightarrow [0, 1]$
- R_i is the reward function for the i^{th} agent, mapping $S \times A \rightarrow \mathbb{R}$

The learning algorithms of the agent must provide it with a policy which will enable the agent to take actions, given the states. The desirable properties of the learning algorithm are *rationality* and *convergence*.

- *rationality* \rightarrow The property that if the policies of other agents converge to a stationary value, then the learning algorithm must converge to a policy which is the best-response to the policy of others.
- *convergence* \rightarrow The learner will necessarily converge to a stationary policy. This property will usually be conditioned on the other agents using an algorithm from some class of learning algorithms.

The learning algorithm which we shall we implementing is **Gra-WoLF** i.e. *gradient based Win or Learn Fast*.

4.4 GraWoLF

GraWolf [6],[2] combines the concepts of **policy gradient learning** and **WoLF variable learning rate**, from the RL domain. Policy Gradient Learning is a technique to handle continuous state space which are often intractable and difficult to work with. WoLF is a multiagent learning technique under which best response learning algorithms to converge in situation of simultaneous learning.

GraWolf, in essence, is based on the idea of “learning quickly” when the agent itself is *losing* and “learning cautiously” when the agent is *winning* [2]. The rationale offered hereby is that the player should try to learn quickly when it is doing more poorly than expected. Similarly, when it is doing better than expected, it should be “cautious” since other players are likely to change their strategies. Whether the player is winning or losing is decided by the following:

- The player is winning if it would consider its present strategy over some equilibrium strategy against the other player’s current strategy. Also,
- The player is considered winning if its expected payoff is currently larger than the value of game’s equilibrium (or equilibria).

4.4.1 Policy Gradient

The policy gradient technique employed hereby is based on the work of Sutton *et al*[2000]. A policy is described by Gibbs distribution over linear combination of agent's state-action pair features. Define:

θ : Policy parameters vector
 ϕ_{sa} : Feature Vector for state s and action a

Then the stochastic policy can be defined according to Gibbs distribution as follows:

$$\pi_{\theta}(s, a) = \frac{e^{\theta \cdot \phi_{sa}}}{\sum_b e^{\theta \cdot \phi_{sb}}} \quad (1)$$

The policy parameters under Gibbs distribution are update according to the following iteration rule (convergence proven by Sutton *et al*):

$$\theta_{k+1} = \theta_k + \delta_k \sum_s \left(d^{\pi_k}(s) \sum_a \phi_{sa} \cdot \pi_{\theta_k}(s, a) \cdot f_{w_k}(s, a) \right) \quad (2)$$

Here $f_{w_k}(s, a)$ is an independent approximation of $Q^{\pi_{\theta_k}}(s, a)$ (with weight parameter vector \mathbf{w}) - which is the expected value of taking action a from state s and then following the policy π_{θ_k} . $d^{\pi_k}(s)$ is state s 's contribution to policy's overall value. The approx. function should carry the form given below:

$$f_{w_k}(s, a) = w_k \cdot \left[\phi_{sa} - \sum_b \pi_{\theta_k}(s, b) \cdot \phi_{sb} \right] \quad (3)$$

This leads to an interesting observation. The same approximation function now carries the form of the *advantage function*,

$$A^{\pi}(s, a) = Q^{\pi}(s, a) - V^{\pi}(s)$$

Here, $V^{\pi}(s)$ is the value of expected discounted reward starting from state s and following policy π thereafter.

On-Line Learning:

It should be noted that since we visit the state on-policy (based on $\pi_{\theta}(s, a)$ values), the contribution from future states is weighed by a discount factor (thereby taking into account their smaller contributions). Therefore, the sum over states in eqn (2) can be replaced by a discount factor (γ), as follows:

Policy Improvement \rightarrow

$$\theta_{k+1} = \theta_k + \delta_k \cdot \gamma^t \cdot \sum_a \phi_{sa} \cdot \pi_{\theta_k}(s, a) \cdot f_{w_k}(s, a) \quad (4)$$

if time t has elapsed since start of the trial.

Since we are using the online policy learning algorithm (“look and update”), policy improvement is immediately followed by value estimation.

Value estimation: SARSA \rightarrow

Value-estimation is performed using the gradient-descent SARSA(λ) algorithm proposed by Richard Sutton and Andrew Barto (1998). If the agent is in state s at time instant k , takes an action a transitioning it to state s' and then it takes action a' , then eligibility trace vector is updated as follows:

$$e_{k+1} = \lambda\gamma^t e_k + \phi_{sa} \quad (5)$$

Weight update \rightarrow

$$w_{k+1} = w_k + e_{k+1}\alpha_k (r + \gamma^t Q_{w_k}(s', a') - \gamma^t Q_{w_k}(s, a)) \quad (6)$$

where λ is known as **SARSA** parameter, $Q_{w_k}(s, a)$ are state-action value function estimates from SARSA and α_k is time-decaying learning rate. This gives:

$$f_{w_k}(s, a) = Q_{w_k}(s, a) - \sum_a \pi_{\theta_k}(s, a) \cdot Q_{w_k}(s, a) \quad (7)$$

4.4.2 Variable Learning Rate (δ_k) Selection: Win or Learn Fast

WoLF (“Win or Learn Fast”) algorithm by Bowling and Veloso encourage convergence in algorithms - guaranteed to find locally optimal policies in non-stationary environments - that don’t converge otherwise. The underlying intuition behind the algorithm is as follows:

- If a learner is doing poorly than expected, then it should learn and adapt itself quickly (in larger steps).
- On the other hand, when it is performing better than expected, it should be *cautious* as other agents might change their policy (sometimes even by the same rule!)

Therefore, rather than trying to resolve the choices/actions of other agents, it tries to incorporate that effect through a variable learning rate. This combination allows the agent to handle the limitations of both itself and that of other agents.

With WoLF, we replace the original learning rate δ_k by two learning rates $\delta_k^w < \delta_k^l$ for winning and losing respectively. The notion of winning or losing is defined using the state-action value function $Q_{w_k}(s, a)$ for current policy parameter vector θ , with the average weight vector over time $\bar{\theta}$. An agent is considered “winning” if and only if:

$$\sum_a \pi_{\theta_k}(s, a) Q_{w_k}(s, a) > \sum_a \pi_{\bar{\theta}_k}(s, a) Q_{w_k}(s, a) \quad (8)$$

The entire learning algorithm can be summarized as follows:

1. Let $\alpha \in (0, 1]$, $\delta^w < \delta^l \in (0, 1]$, $\beta \in (0, 1]$ be learning rates. Initialize:

$$w \leftarrow \bar{0}, \theta \leftarrow \bar{0}, \bar{\theta} \leftarrow \bar{0}, e \leftarrow \bar{0}$$

2. Define:

$$\begin{aligned}\pi_\theta(s, a) &= \frac{e^{\theta \cdot \phi_{sa}}}{\sum_b e^{\theta \cdot \phi_{sb}}} \\ Q_w(s, a) &= w \cdot \phi_{sa} \\ f_{w_k}(s, a) &= w_k \cdot [\phi_{sa} - \sum_b \pi_{\theta_k}(s, b) \cdot \phi_{sb}]\end{aligned}$$

3. Repeat:

- (a) From state s select action a according to strategy $\pi^\theta(s)$
- (b) Observed reward = r and next state = s' with time elapsed = t ,

$$\begin{aligned}e_{k+1} &= \lambda \gamma^t e_k + \phi_{sa} \\ w_{k+1} &= w_k + e_{k+1} \alpha_k (r + \gamma^t Q_{w_k}(s', a') - \gamma^t Q_{w_k}(s, a)) \\ \theta_{k+1} &= \theta_k + \delta_k \cdot \gamma^t \cdot \sum_a \phi_{sa} \cdot \pi_{\theta_k}(s, a) \cdot f_{w_k}(s, a)\end{aligned}$$

where,

$$\delta = \begin{cases} \delta^w, & \text{if } \sum_a \pi_{\theta_k}(s, a) Q_{w_k}(s, a) > \sum_a \pi_{\bar{\theta}_k}(s, a) Q_{w_k}(s, a) \\ \delta^l, & \text{otherwise} \end{cases}$$

- (c) Maintain an average parameter vector,

$$\bar{\theta} \leftarrow (1 - \beta) \bar{\theta} + \beta \theta$$

- (d) If s' is s_0 , or trial is over then $t \leftarrow 0$, $e \leftarrow \bar{0}$. Otherwise $t \leftarrow t+1$.

4.4.3 Important Observations

- Gibbs distribution over the linear combination of feature elements and gradient-descent SARSA (λ) can be thought of as actor and critic respectively in an **actor-critic** paradigm setting. The WoLF principle takes care of how the actor adjusts its parameters based on critic's response.
- The feature vector ϕ_{sa} is the only context-dependent (or task dependent) parameter in the entire learning algorithm, apart from reward r . Defining feature vector refers to approximating and generalizing the state-action space. One method of generating the feature vector is tile coding, which is described next.

4.5 Tile Coding

Tile Coding [1] is popular technique for handling intractable feature spaces but creating a set of boolean feature from a set of continuous features. In tile coding, the feature vector fields are grouped into input space partitions which are exhaustive but not necessarily mutually exclusive. Each such partition is called a *tiling*, and each element of the partition is called a *tile*. Each tile represents one binary feature.

By laying offset tilings over a given multidimensional continuous feature space, each point in continuous space now belongs to exactly ONE tile per offset tiling. Since, there is a binary value associated with each tile, a continuous feature space can be represented by a multidimensional boolean vector. Besides, neighboring points will generally fall into the same tile, allowing one boolean value to represent a section of the continuous feature space.

An example of multiple, overlapping grid tilings is shown below:

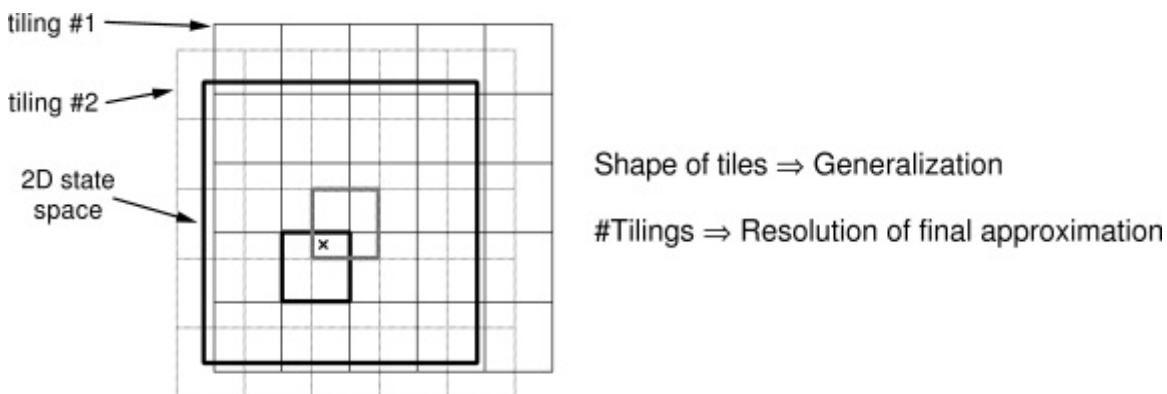


Figure 2: Tile Coding (Image source [1])

The size and shape of tiles are chosen to match the degree of generalization one intends to introduce in the feature space. A greater density of tilings leads to more accurate approximations for desired function, but also add to the computational overheads.

5 Methodology Adopted

We intend to do both physical runs and simulations in our work. The physical implementation will help us gauge the performance in agents with limitations. The simulations will enable us to run the algorithm for numerous agents.

5.1 System setup

The system comprises of an arena with an overhead web camera (Fig. 3). The arena has a circular target zone marked on it. The overhead camera acts as the global input source for the system. The video stream from the camera is provided to the **Image Processing Module** which tracks the robots, determines the state variables of the environment (position, velocity, acceleration of robots) and provides these as input to the **Learning Module**. Depending on the current policy, an action is decided and the policy is updated. The **Actuation Module** then converts these actions into commands for the robots and communicates it to the robots via Bluetooth.

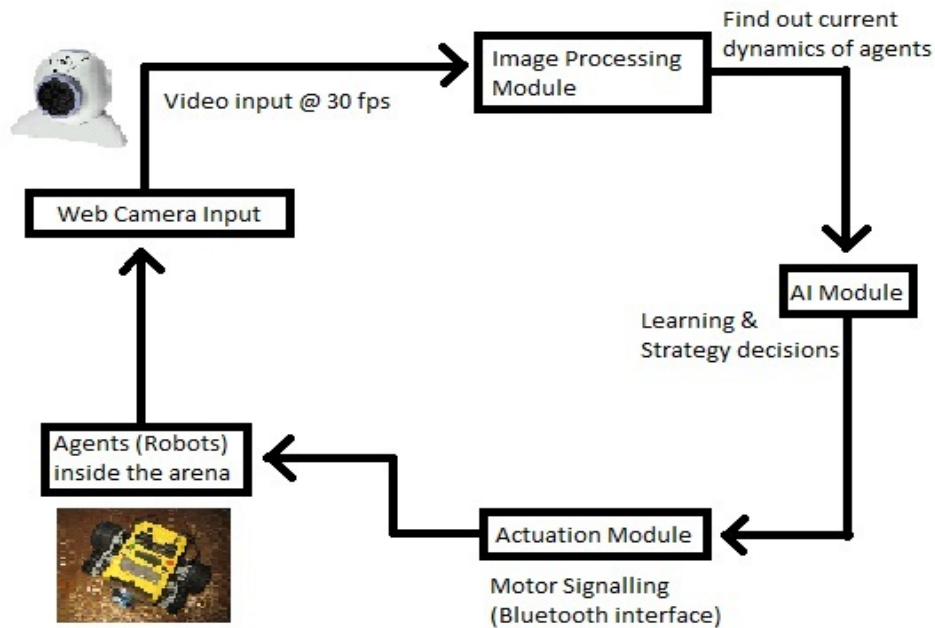


Figure 3: Loop depicting flow of information in the system

5.2 Proposed Reward Function Formulation

We model the robot learning strategies based on the following two heuristics:

- The attacker should try to move as *close* to the ball and in the *least* amount of time.
- The defender should try to keep the attacker away from the ball for *as long as possible*.

Before assigning the rewards and penalties for individual robots, we define following two functions:

- **Distance Potential Function**, $f(r)$ \rightarrow The reward function for the attacker that increases with decreasing distance of the attacker from the ball.
- **Timing Potential Function**, $g(t)$ \rightarrow The penalty function for the attacker that increases with increasing time taken by the attacker to reach the ball.

The exact form of these functions shall be discussed later.

5.2.1 Rewards/Penalties for Attacker

- The attacker is assigned a reward R_0 for successfully accomplishing the task of reaching the ball within the given time frame. Similarly, an equal penalty is inflicted in case the task is not carried out.
- A reward $f(r)$ is assigned to the attacker, depending on its distance from the ball. This motivates the attacker to move as close to the ball as possible.
- A penalty $g(t)$ is inflicted on the attacker, depending on the time it takes to accomplish the task. The more the time, the more is the penalty. This urges the attacker to finish the job in least time.

So, the attacker reward function can be written as:

$$R_A = \pm R_0 + f(r) - g(t) \quad (9)$$

5.2.2 Rewards/Penalties for Defender

- The defender is assigned a reward R_0 for successfully accomplishing the task of preventing the attacker from reaching the ball within the given time frame. Similarly, an equal penalty is inflicted in case the task is not carried out.
- A penalty $f(r)$ is inflicted on the defender, depending on attacker's distance from the ball. This motivates the defender to keep the attacker as far away from the ball as possible.
- A reward $g(t)$ is assigned to the defender, depending on the time for which it successfully keeps the attacker away from the ball. The more the time, the more is the reward.

So, the defender reward function can be written as:

$$R_D = \pm R_0 - f(r) + g(t) \quad (10)$$

6 Robot Simulation

The simulation for the adversarial task at hand was carried out with the help of SDL Libraries (C++) in Microsoft Visual Studio 2008 IDE. The current simulation setup had been developed for *one-on-one* adversarial task of Keepout. The Fig. 4 shows a snapshot of the simulation window:

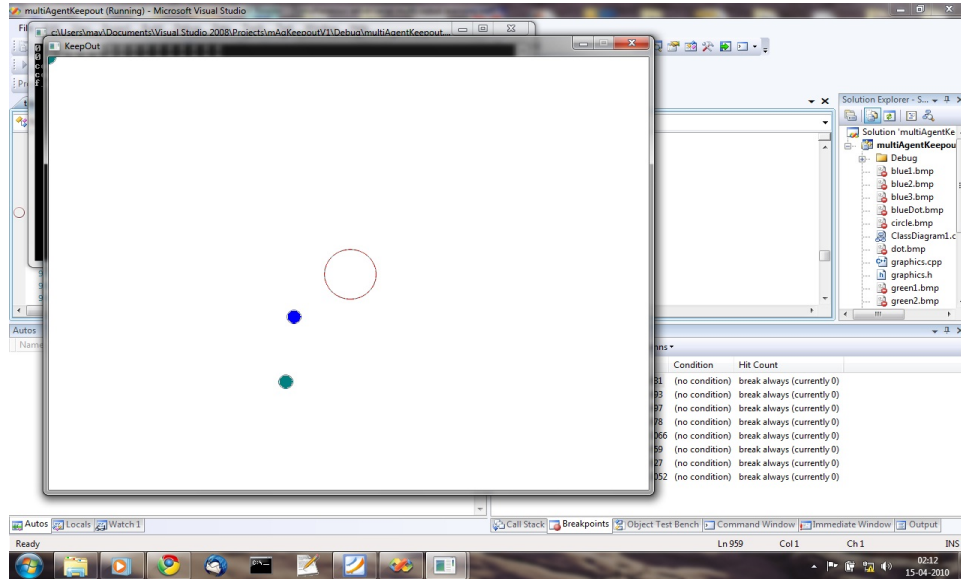


Figure 4: Simulation Window snapshot

The green dot stands for the attacker robot, while the blue one corresponds to the defender. A delay of 10 ms is put between two consecutive instructions to either agents. This is an attempt towards simulating the *latency* factor that is normally associated with real robots.

6.1 Implementation Details

We consider a seven-dimensional state space for our implementation, with 16 tilings. Therefore, we work with a 16 dimensional feature array representing the seven dimensional feature space. Various elements of this space are described below:

- State (s): The seven dimensional state vector is described as:
 1. Radial distance of attacker
 2. Radial distance of defender
 3. Angle between them
 4. Attacker velocity magnitude
 5. Attacker Velocity angle wrt radial line
 6. Defender Velocity magnitude
 7. Defender Velocity angle wrt radial line

- Action (a): The action space is simplified by making the attacker choose action points along a line passing through the target center and perpendicular to line joining the attacker to target center, thrice the length of the latter. This line is divided into seven equidistant points, one of which is chosen at each moment by the attacker as the action point. The defender uses the same points, but it navigates to a point bisecting the chosen action point and attacker's position. (shown in Fig. 5 below)
- State-Action (s, a): The state-action space is eight dimensional, with the chosen action point as the eighth dimension, apart from the seven dimensional state vector.
- Tiling: We use the CMACS tile coding [1] for our simulation to construct the feature vector. Sixteen tilings with discretizations of 800 mm for distance measures, 1600 mm/s for velocity measures and 20 degrees for angles have been employed.
- Learning rate parameters: The learning rate parameters have been described in Appendix.
- Reward: A reward of ± 1 is used for attacker/defender depending on who wins the task. It should be noted that this is the only reward that is associated with Bowling and Veloso's implementation of GraWolf. We have tried to propose and test the reward structure mentioned under Sec. 5.2.

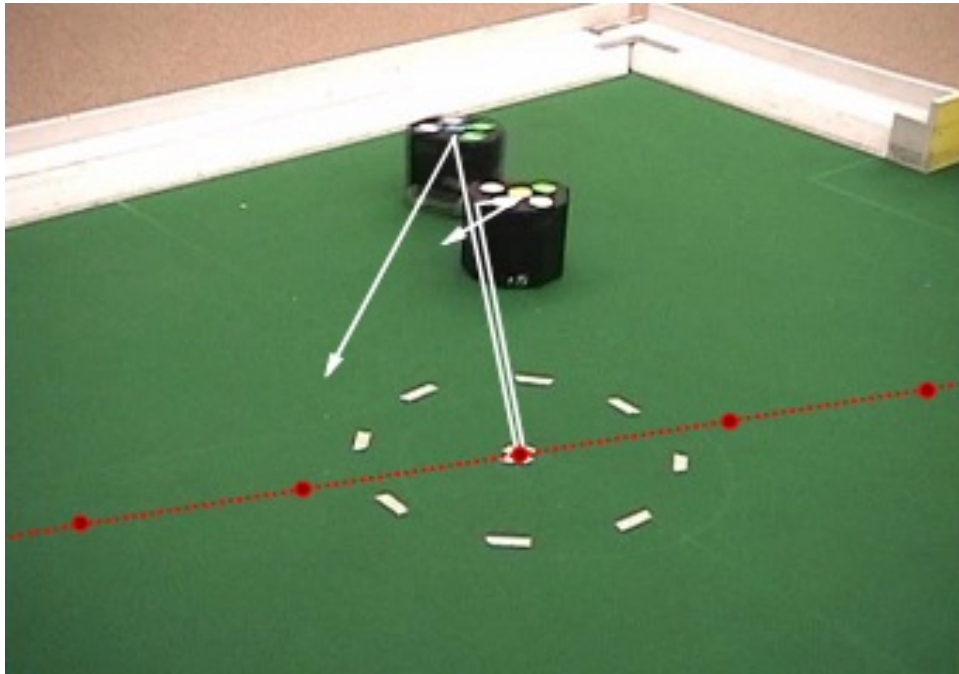


Figure 5: Action space for the given problem. White lines depicting agents' distance from the target and direction of heading are shown. Five of the seven action points have been shown on the broken red line passing through target point. The robots shown belong to CMDragons'02 Robot Soccer team (Image source [2])

7 Results

7.1 Learning v/s Random Trials

We choose the CMAES tiling scheme [1] with 16 Tilings and 64 hash bins. Value estimation learning rate for attacker is $\alpha = 0.002$ and that for defender as mentioned in Appendix. We obtained the following wins-losses table for the attacker and defender over 150 runs of the game:

Attacker-Defender	Attacker Wins	Defender Wins
L v/s L	0.42	0.58
L v/s R	0.48	0.52
R v/s L	0.39	0.61

Table 1: Keepout Performance Table

Here L = Learning agent , R = Random behavior.

Now, the reward function was set in accordance with our proposed form, i.e.

$$R_A = \pm R_0 + f(r) - g(t)$$

for the attacker, and its negative for the defender. The functions were chosen as below:

- $f(r) = (\text{iniAttackerDist} - \text{currAttackerDist}) / \text{iniAttackerDist}$
- $g(t) = t / \text{maxTrials}$

Here,

iniAttackerDist = initial distance of attacker from target point

currAttackerDist = current distance of attacker from target point

maxTrials = T_0

We also set the reward function according to the proposed solution heuristic and conduct a run of 100 trials with both Attacker and Defender learning. It yielded a 38:62 win ratio in favor of defender.

7.2 Comparing Value estimation learning rates α

The speed of attacker bot is set to 15 while that of defender bot is 10 units. This is done in order to offer the attacker a fair chance, as it was realized through simulations that the defender was able to win most of the trials just by standing in the path of the attacker and not moving much. Each time a sequence of 150 trials is conducted.

The value estimation learning rates were chosen to be of the following form:

$$\alpha_A = \alpha_a * 0.00001 / (1 + t / 200000)$$

for the attacker, and

$$\alpha_D = \alpha_b * 0.1 / (1 + t / 200000)$$

for the defender.

The following graph was obtained from this experiment:

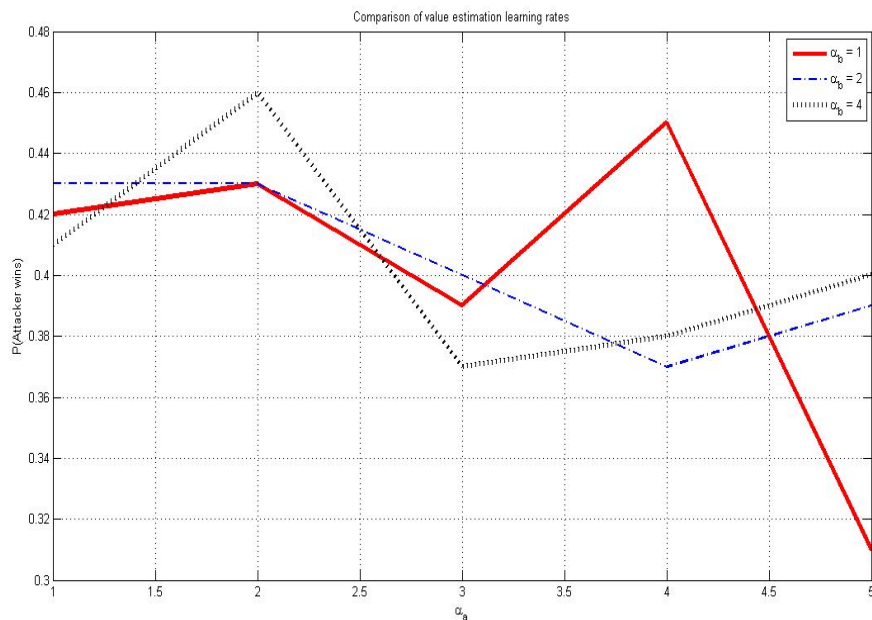


Figure 6: Comparing Value estimation learning rates α

7.3 Comparing Attacker and Defender Speeds

The value estimation rates were chosen to be:

$$\alpha_A = 0.00004/(1 + t/200000)$$

$$\alpha_D = 0.2/(1 + t/200000)$$

The attacker and defender speeds were chosen to be of the following form:

attackerDot.speed = A, defenderDot.speed= 5 * B

Once again, the proposed reward function formulation was employed.

The following graph was obtained from this experiment:

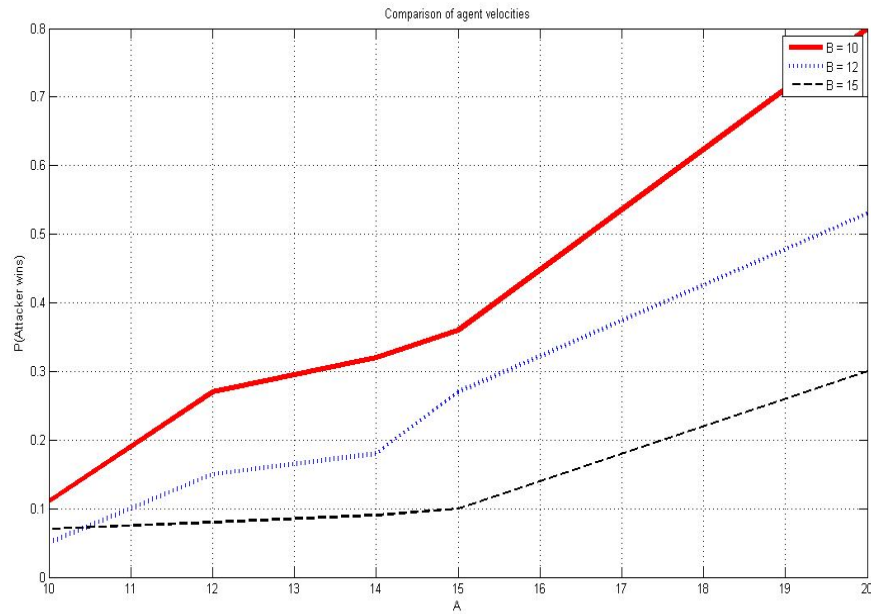


Figure 7: Comparing attacker and defender speeds

7.4 Comparing Angular separation and agent velocities

Choice of parameters:

- attackerDot.speed= 15, defenderDot.speed= 10
- $\alpha_A = 0.00002 / (1 + t / 200000)$
- $\alpha_D = 0.4 / (1 + t / 200000)$
- Number of episodes = 100

The following graphs were obtained from this experiment:

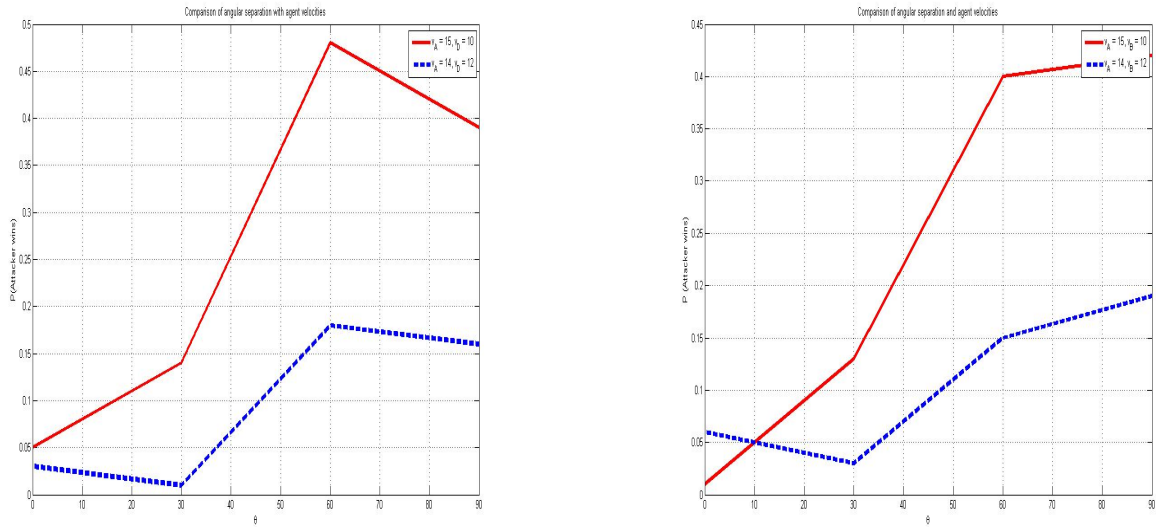


Figure 8: Comparing Angular separation and agent velocities for (a) rewards = R_0 and (b) our reward formulation

8 Conclusions

We can draw the following conclusions from the above set of results:

- Sec 7.1 : Learning agents far outperform those that make random action selection. This accentuates the importance of learning in a stochastic multirobot setting, besides stressing on the importance of GraWoLF algorithm as a learning paradigm. We also realize that the formulation of the proposed reward function tends to favor the defender in this particular case.
- Sec 7.2 : Neither agent's performance is directly related to increase in its value estimation learning rate. This points to the fact that plainly increasing this rate might not essentially improve the performance of the agent over an extended sequence of trials.
- Sec 7.3 : This section provides a straightforward and much logical explanation of the fact that increase in attacker's speed MUST be reciprocated by a concomitant increase in defender's speed. Obviously, to chase a fast moving agent, another quick agent is required.
- Sec 7.4 : This experiment provides two important insights:
 1. A decrease in the relative speeds of the agents tends to favor the defender over a wide range of initial angular separations.
 2. For a larger value of initial angular separation, the attacker tends to gain from our reward function formulation in comparison to that of [2].

In essence, we can conclude that learning agents are far superior in task realization as compared to random agents. Also, various experiments with different learning parameters and initial settings provide us some very basic and logical conclusion, thereby stressing on the genuinity of our reward function formulation and the learning scheme as a whole.

9 Scope for future work

We have been able to achieve the working simulation for a 2-agent problem scenario for the given task in the stipulated time. This leaves us with two important achievements to still lookout for:

1. Trying to shift onto a 2 – *on* – 2 scenario, with two teams of two robots each, whereby the action of each agent will depend on not one, but three other agents. This scenario will be subjected to various tests, with & without the application of cooperation, as well as different learning strategies.
2. Implement the learning algorithm hence developed on a team of real Lego robots. Such implementation will not only test the practical usefulness of this algorithm, but also put a whole lot of new real-world challenges like latency, partial observability (leading to POMDPs) etc.

If the above tasks are achieved to a fair extent, the following additional plans can be interesting to carry out:

- Some obstacles such as walls etc. can be planted on the arena, to restrict the motion of agents and invoke the need of path planning. It would be intriguing to observe how robots cooperate in a cluttered adversarial setting.

References

- [1] R. S. Sutton and A. G. Barto, “Reinforcement learning: Introduction,” 1998.
- [2] M. Bowling and M. Veloso, “Simultaneous adversarial multi-robot learning,” Tech. Rep., 2003.
- [3] M. Tan, “Multi-agent reinforcement learning: Independent vs. cooperative agents,” in *In Proceedings of the Tenth International Conference on Machine Learning*. Morgan Kaufmann, 1993, pp. 330–337.
- [4] M. Bowling and M. Veloso, “An analysis of stochastic game theory for multiagent reinforcement learning,” Computer Science Department, Carnegie Mellon University, Tech. Rep., 2000.
- [5] M. Lauer and M. Riedmiller, “Reinforcement learning for stochastic cooperative multi-agent-systems,” in *In Proceedings of the AAMAS 04*, 2004, pp. 1516–1517.
- [6] M. Bowling, “Multiagent learning in the presence of agents with limitations,” Ph.D. dissertation, Pittsburgh, PA, USA, 2003, chair- Veloso, Manuela.

Appendix

Learning Rate Parameters

- Discount factor, $\gamma = 0.8$
- SARSA learning rate, $\lambda = 0.5$
- Learning Rate during winning, $\delta^w = 0.2$
- Learning Rate during losing, $\delta^l = 0.4$
- Decaying learning rate, $\alpha_k = \frac{0.1}{1 + \frac{t}{200000}}$

Class Diagram

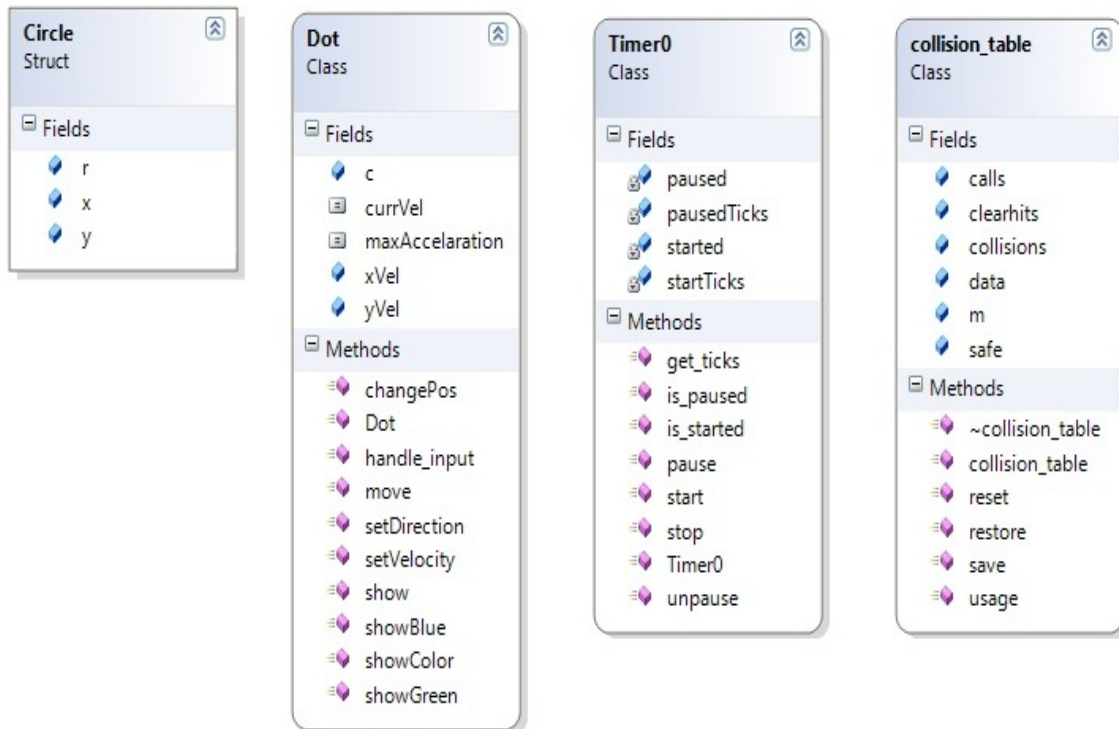


Figure 9: Class Diagram for 2 agent GRAWoLF Implementation

Working with Lego Robots

Robots

The robots were constructed using the Lego Mindstorms NXT 2.0 Kits. Lego kits offer us the advantage of building robots with ease, besides having built-in wireless (Bluetooth) communication abilities.

Software Modules

The different modules and dependencies involved are described below:

1. **Image Processing Module** → This module was developed using Intel’s Open Computer Vision (OpenCV) library. It is used to process the input video from the camera and determines the different state variables of the environment (Fig. 10).

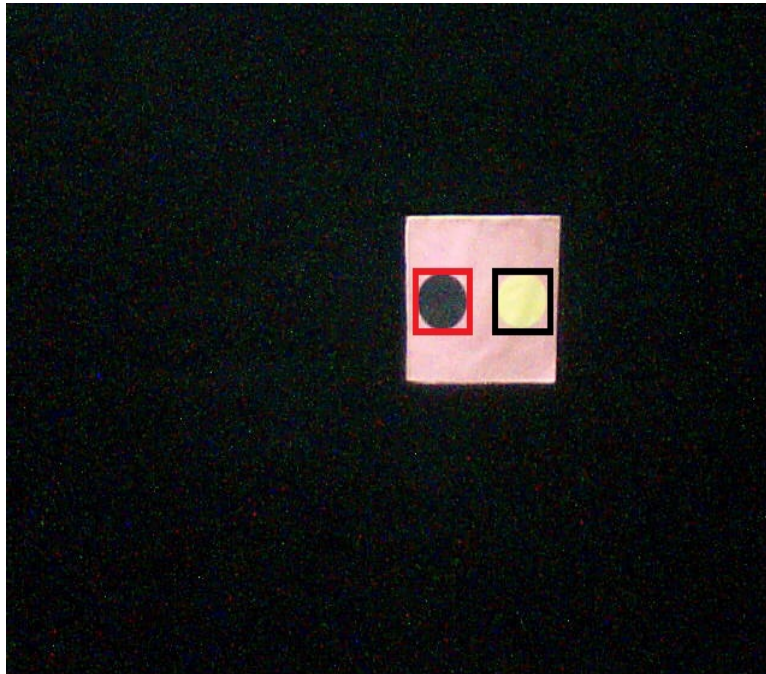


Figure 10: Finding out robot position and velocity with the help of overhead tag. Given view is from the overhead camera.

2. **Learning Module** → This module is an implementation of the GraWoLF algorithm for team of agents (in C++), developed currently for one-on-one setting. The class diagram for our implementation is shown in Fig. 9 in Appendix.
3. **Actuation Module** → The actuation module was developed using NXT++ Lego Interfacing library. NXT++ is an interface written in C++ that allows the control LEGO MINDSTORMS NXT robots directly through a USB or Bluetooth connection.