# Multi-Agent Cooperative Pursuit Evasion Game

*Thesis submitted by*

**Raghav Gupta 2017EE10544**
**Sarthak Garg 2017EE30546**

*Under the guidance of*

**Dr. Deepak U. Patil**

*in partial fulfilment of the requirements*
*for the award of the degree of*

**BACHELOR OF TECHNOLOGY**

**Department of Electrical Engineering,**
**Indian Institute of Technology, Delhi**
**January 2021**

# Certificate

This is to certify that the thesis titled **Multi-Agent Pursuit Evasion Games**, submitted by **Sarthak Garg and Raghav Gupta**, to the Indian Institute of Technology, Delhi, for the award of the degree of **Bachelor of Technology**, is a bona fide record of the research work done by them under our supervision. The contents of this thesis, in full or in parts, have not been submitted to any other Institute or University for the award of any degree or diploma.

**Deepak U. Patil**
**Department of Electrical Engineering**
**Indian Institute of Technology, Delhi**

# Acknowledgments

We would like to express our sincere gratitude to our supervisor Dr. Deepak U. Patil for providing us the opportunity to work on this project. The valuable advice and constant support he provided helped us to progress through the project smoothly.

We would also like to thank our families and friends for supporting us throughout the project.

**Raghav Gupta**
**Sarthak Garg**

# Abstract

Pursuit-Evasion is a classic problem in the field of Mathematics and Control Theory where a group of agents search another group of agents in an environment. This has numerous examples and applications in real-world situations. Reinforcement Learning is an extension to classical machine learning and has gained pace in the recent years. Reinforcement Learning algorithms have been applied for the pursuit evasion tasks in the past to to make the agents develop strategies in different situations.

Though a powerful approach, standard Reinforcement Learning approaches like Q-learning or Monte-Carlo methods cannot be applied to practical applications due to the high computational requirements and exploding state-action spaces. We explore the use of Deep Q-Networks, using deep learning, to optimise the reinforcement learning algorithms and reduce complexity. A lot of related work in this field focuses on a few agents. We have explored combining Deep Q-Networks and Mean Field Theory to allow for a large number of agents by approximating multiple agents as a single agent. We have carried out detailed comparisons between the results of our approach to two other standard approaches applied to the same task.

# Contents

**1 Introduction** **1**

1.1 Pursuit-Evasion Game . . . 1

1.2 Objective . . . 1

**2 Reinforcement Learning** **3**

2.1 Introduction . . . 3

2.2 Finite Markov Decision Process . . . 4

2.3 Q-Learning . . . 6

**3 Literature Survey** **8**

3.1 Related Work . . . 8

3.2 Deep Q-Networks . . . 9

3.2.1 Target Network . . . 10

3.2.2 Experience Replay . . . 11

3.3 $\epsilon$-Greedy Approach . . . 12

3.4 Mean Field Reinforcement Learning . . . 12

**4 Problem Formulation and Solving Approach** **15**

4.1 Game Setting . . . 15

4.2 Pursuit Evasion game modelled as a reinforcement learning problem . . . 15

4.3 Reward Structure . . . 16

4.3.1 Naive Reward Strategy . . . 17

4.3.2 Optimal Reward Strategy . . . 17

4.4 Implementation of three approaches . . . 18

4.4.1 Independent Agents Approach . . . 18

4.4.2 Full State Space Approach . . . 18

4.4.3 Mean Field Approach . . . 19

4.5 Deep Q-Networks Implementation . . . 20

**5 Implementation and Code** **22**

5.1 Dependencies . . . . . . . . . . . . . . . . . . . . . . . . . . . . 22
5.2 Pseudo-Code . . . . . . . . . . . . . . . . . . . . . . . . . . . . 23
5.3 MAgent Framework . . . . . . . . . . . . . . . . . . . . . . . . . 24
5.4 Code . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 24

**6 Experiments and Results 27**
6.1 Selecting Parameters . . . . . . . . . . . . . . . . . . . . . . . . 27
6.2 Base Experiment: 1 Pursuer vs 1 Evader . . . . . . . . . . . . . . 27
6.3 Experiment: Multiple Pursuers vs 1 Evader . . . . . . . . . . . . 30
6.4 Comparison of performance between the three approaches . . . . . . 34
6.4.1 Training performance . . . . . . . . . . . . . . . . . . . . . 34
6.4.2 Communication between agents . . . . . . . . . . . . . . . . . 35
6.4.3 Absolute performance on testing . . . . . . . . . . . . . . . . 36
6.4.4 Time taken to catch the evader . . . . . . . . . . . . . . . . 36
6.4.5 Computation time . . . . . . . . . . . . . . . . . . . . . . . 37
6.4.6 Conclusion: Mean Field is the winning approach . . . . . . . . 38
6.5 Quantifying Cooperation . . . . . . . . . . . . . . . . . . . . . . 38

**7 Conclusion and Future Work 41**
7.1 Summary . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 41
7.2 Future Work . . . . . . . . . . . . . . . . . . . . . . . . . . . . 41

**References 43**

# List of Figures

2.1 Agent-Environment Interaction . . . . . . . . . . . . . . . . . . . 4
2.2 Q-Learning algorithm . . . . . . . . . . . . . . . . . . . . . . . 7

3.1 Deep Q-Network Schematic . . . . . . . . . . . . . . . . . . . . . 11
3.2 Interaction between single agent and average effect of neighbours . . . . . 13

4.1 Game Setting with 1 Pursuer and 1 Evader . . . . . . . . . . . . . 16
4.2 DQN Implementation for three approaches . . . . . . . . . . . . . 20

6.1 Selection of parameters for the experiments . . . . . . . . . . . . . . . . . 27
6.2 Pursuer rewards and losses . . . . . . . . . . . . . . . . . . . . . . . . 28
6.3 Evader rewards and losses . . . . . . . . . . . . . . . . . . . . . . . . 28
6.4 Path Trace [Blue: Pursuer and Red: Evader] . . . . . . . . . . . . . . . . 29
6.5 Training Rewards and Losses for Pursuers . . . . . . . . . . . . . . . . . 31
6.6 Pursuer Violations . . . . . . . . . . . . . . . . . . . . . . . . . . . . 32
6.7 Snapshots of simulation View GIFs on Github . . . . . . . . . . . . 33
(a) GIF1: Full State . . . . . . . . . . . . . . . . . . . . . . . . . . . . 33
(b) GIF2: Full State . . . . . . . . . . . . . . . . . . . . . . . . . . . . 33
(c) GIF1: Independent . . . . . . . . . . . . . . . . . . . . . . . . . . . 33
(d) GIF2: Independent . . . . . . . . . . . . . . . . . . . . . . . . . . . 33
(e) GIF1: Mean Field . . . . . . . . . . . . . . . . . . . . . . . . . . . 33
(f) GIF2: Mean Field . . . . . . . . . . . . . . . . . . . . . . . . . . . 33
6.8 Comparison between average pursuer training rewards (1 epoch = 10 episodes) 34
6.9 Comparison between average pursuer DQN loss (1 epoch = 10 episodes) . 35
6.10 Probability of catching the evader vs max. number of time steps allowed 37
6.11 Time taken for Q-function estimation as number of pursuers are varied . 38
6.12 Reward share between pursuers during training . . . . . . . . . . . . . . 39
(a) Initial Stage . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 39
(b) Intermediate Stage . . . . . . . . . . . . . . . . . . . . . . . . . . . 39
(c) Final Stage . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 39
6.13 Cooperation Metric during three phases of training . . . . . . . . . . . . 39

# List of Tables

6.1 Total messages exchanges between the four pursuers (Training) . . . . . . 35
6.2 Number of successful episodes during testing . . . . . . . . . . . . . . . . 36

# Chapter 1

# Introduction

## 1.1 Pursuit-Evasion Game

Pursuit-Evasion is a famous problem in Control Theory and Mathematics. It involves formulating strategies for a set of agents to capture another set of agents or resist capture from another set of agents in an environment. The group of agents trying to catch are termed as *pursuers* and the agents which are trying to evade from *pursuers* are called *evaders*. Thus, the aim of pursuers is to catch evaders as quickly as possible and on the other hand evaders need to outrun the pursuers for as long as possible. The game could be set up in an environment which could be either Discrete or Continuous. The catching criteria may vary in different variants, like occupying same position or coming within a finite range of distance. In case of multiple agents, this problem becomes challenging since agents have to develop cooperative strategies in order to succeed. Moreover in large state space, scaling of the system becomes a challenge as computation resources are limited.

The solution to this problem can be utilized in various practical scenarios like:

- **Missile Guidance Systems:** Targets could be modelled as evaders and missiles can be modelled as pursuers. Intelligent actions based on the adversary moves could be utilized in such systems.
- **Search and Rescue Missions:** Pursuers aim to catch fixed evader and this could be modelled as robotic search operations.
- **Patrolling Systems:** Multiple pursuers (cops) can try to formulate cooperative strategies against evaders (robbers).

## 1.2 Objective

A similar project was attempted by our senior Yash Garg (Electrical Department, IIT Delhi) for his Bachelor Thesis, which involved solving this problem in Discrete Space for a maximum of 2 agents in the environment on one side using Reinforcement Learning.

We are extending that work to the case of multiple agents in Discrete Space in order to study the performance of agents using different approaches and quantify and compare cooperation between agents in various settings. Reinforcement Learning is a popular and effective way for Multi-agent RL problems. However, traditional Reinforcement Learning techniques do not scale well with increased agents in the environment. Moreover, providing cooperative abilities to multiple agents in an RL problem cannot be achieved with straightforward approaches.

We aim to do the following for our project:

1. Set up a pursuit evasion game for multiple agents using reinforcement learning, where a group of pursuers try to catch an evader agent in discrete space.
2. Inspect and evaluate the performance and emergent cooperation of pursuer agents on varying the training approach and find out a scalable solution for large number of agents.

# Chapter 2

# Reinforcement Learning

Our work deals with Reinforcement Learning (RL) to train the agents and then study their behaviour in different situations. This chapter briefly discusses the relevant concepts and basic terminologies of Reinforcement Learning used throughout the course of this project.

## 2.1 Introduction

Reinforcement Learning deals with training agents to make a sequence of decisions to reach their goal. Essentially, it maps situations to actions in order to maximize a numerical reward received from the environment at every step. Reinforcement Learning problems are in a way closed-loop problems as the actions taken by the system undergoing training influences its later inputs and thus, are closely related to Optimal Control Theory. It is one of the areas in the field of Machine Learning (Supervised/Unsupervised). As RL does not require training sets (labelled data) to be given in order to predict in unseen data, it is different from Supervised Learning. It is also different from unsupervised learning, where one tries to find patterns in unlabeled data in order to label them correctly.

The basic terminologies in the field of RL are given as follows:

- **Policy:** It is the decision-making function of agent. Precisely, it maps the states in an environment to the actions that should be taken in those states. It could be referred to as the 'Control-Strategy' of agent. Policies could either be deterministic (clearly defined actions for every state) or stochastic (probability distribution of actions for every state).

- **Reward:** The numeric signal received by agents from the environment after taking an action. The goal of agent is to obtain high reward over long run.

- **Value Function:** Value Function of a state represents the future reward obtained, starting from that state, based on a given policy. Rewards measure the immediate desirability of states whereas Value Function represents the long-run desirability of a state.

- **Model:** Model of the environment allows it to predict the future states and rewards, starting from some state. So, models can be used to predict the possible future states without even experiencing them and thus, analyzing the course of action. Methods that make use of such strategy (planning) are called model-based methods and methods which do not are called model-free methods.

## 2.2 Finite Markov Decision Process

Reinforcement Learning problems involve training an agent to maximize its reward earned in long run through its course of action to achieve a goal. The decision maker i.e. agent, in a particular state $S_t$ of the environment takes an action $A_t$. The environment responds back with the next state $S_{t+1}$ and the reward obtained $R_{t+1}$ for that action. The agent takes action using its policy $\pi_t$, where $\pi_t(a|s)$ gives the conditional probability of taking an action $a$ in a state $s$. This is the basic interaction that lies at the heart of Reinforcement Learning and is depicted by Figure 2.1 [1].

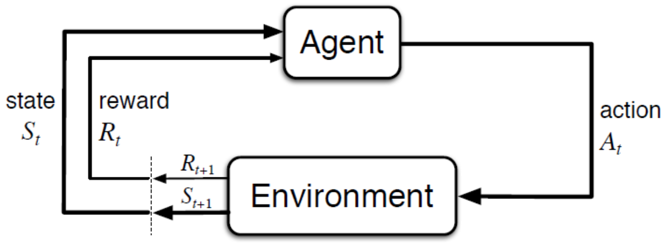


Figure 2.1: Agent-Environment Interaction

We know the objective of RL problem i.e. agents have to obtain maximum reward over long run. If rewards earned by an agent after time step t are $R_{t+1}, R_{t+2}, R_{t+3}...$, then the long term reward obtained by agent or *return* ($G_t$) can be written as:

$$G_t = R_{t+1} + R_{t+2} + R_{t+3} + R_{t+4} + ..... + R_T$$

Here $T$ is the final time step signifying that the task is *episodic*, meaning that the agent-environment interaction breaks on terminal states as opposed to *continuing* tasks where there is no terminal state. The aim of the agent is to maximize the *expected return*.

Another important parameter is the *discount rate* ($\gamma$), which helps to determine the present value of future rewards. Thus, a reward earned $n$ time steps in future would be discounted by multiplying with $\gamma^{n-1}$. Therefore, the *return* ($G_t$) is given by:

$$G_t = R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \gamma^3 R_{t+4} + ...$$

Here $0 \leq \gamma \leq 1$. If discount rate is 0, then we can say that the agent only values the immediate reward and its goal is to maximize the immediate reward earned. On the other hand discount rate of 1 implies that the weightage to future rewards is much stronger and the agent is more farsighted.

### Markov Property

At any time step $t$, the environment responds back with next state $S_{t+1}$ and reward obtained $R_{t+1}$. This response can be modelled as a probability distribution given by Equation 2.1.

$$Pr\{S_{t+1} = s', R_{t+1} = r | S_0, A_0, R_1, ....S_{t-1}, A_{t-1}, R_t, S_t, A_t\} \tag{2.1}$$

A state is said to have Markov property if Equation 2.1 could be simplified as Equation 2.2 meaning that knowledge of just previous state, action and reward is equivalent to the complete history of these variables. A Reinforcement Learning task that satisfies the Markov Property is said to be a *Markov Decision Process* or *MDP*. All of our work is based on finite MDPs.

$$p(s', r|s, a) = Pr\{S_{t+1} = s', R_{t+1} = r | S_t = s, A_t = a\} \tag{2.2}$$

### Bellman Equation

As discussed in this chapters beginning, Value Function is an important aspect of RL. There are two kinds of Value Functions in literature: State-Value Function and Action-Value Function. State-Value Function for a policy $\pi$, represented by $v_\pi(s)$, is the expected return when starting in a state $s$ following policy $\pi$ given by Equation 2.3. Similarly, Action-Value Function represented by $q_\pi(s, a)$ is the expected return from a state $s$ and taking an action $a$ given by Equation 2.4.

$$v_\pi(s) = E_\pi[G_t | S_t = s] \tag{2.3}$$

$$q_\pi(s, a) = E_\pi[G_t | S_t = s, A_t = a] \tag{2.4}$$

Both of these Value Functions satisfy the famous Bellman equations which break these into two parts: the immediate reward earned and the discounted future values. The Bellman equation for State-Value function is given by Equation 2.5 and Bellman equation for Action-Value Function is given by Equation 2.6.

$$v_\pi(s) = E_\pi[R_{t+1} + \gamma v_\pi(S_{t+1})|S_t = s] \tag{2.5}$$

$$q_\pi(s,a) = E_\pi[R_{t+1} + \gamma q_\pi(S_{t+1}, A_{t+1})|S_t = s, A_t = a] \tag{2.6}$$

A key piece of solving an RL problem is finding a policy $\pi$ which maximizes the agents' reward over long run. A policy $\pi$ is said to be better than a policy $\pi'$ if state-value function for each state is better with policy $\pi$ as compared to policy $\pi'$. This means that $v_\pi(s) \geq v_{\pi'}(s)\ \forall\ s \in \mathcal{S}$. In case of an MDP, there always exists an optimal policy $\pi_*$ such that $\pi_* \geq \pi\ \forall\ \pi$. Thus the goal is to find a policy for which action-value function is maximum for each action pair compared to all other policies.

The state-value function and action-value function for the optimal policy $\pi_*$ satisfy the Bellman optimality equations given by Equation 2.7 and Equation 2.8 respectively.

$$v_*(s) = \max_a \sum_{s',r} p(s',r|s,a)[r + \gamma v_*(s')] \tag{2.7}$$

$$q_*(s,a) = \sum_{s',r} p(s',r|s,a)[r + \gamma \max_{a'} q_*(s',a')] \tag{2.8}$$

Solving the Bellman optimality equation gives us $v_*$ and $q_*$ through which we can get the optimal policy. If the system has $N$ states, then $N$ equations need to be solved given that we know the dynamics of the environment $(p(s',r|s,a))$. This is often computationally expensive and the dynamics of the environment are not known most of the times. Thus, simply solving the Bellman optimality equations is not a solution for practical problems.

## 2.3 Q-Learning

Q-Learning, as the name suggests, is an algorithm to learn the action-values function $(q(s,a))$ and select actions in different states appropriately. It is a *model-free* RL algorithm meaning that it does not have a model that mimics the environment or predicts future states and rewards without even experiencing them. Learning facilitated through this algorithm is *off-policy learning* implying that the agents do not improve the policy that they follow during training. Instead, as the training progresses, the optimal policy keeps on improving itself through repeated iterations.

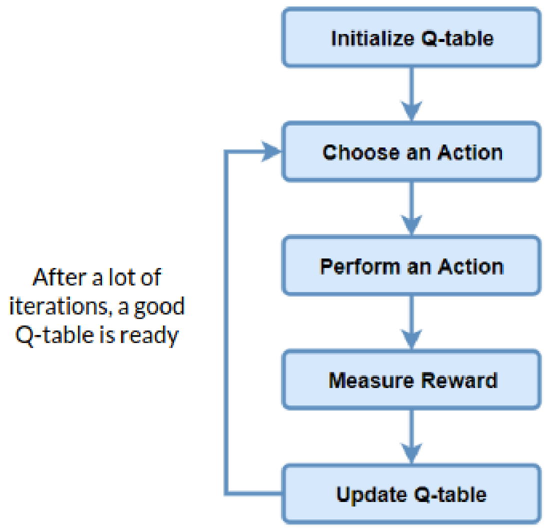


Figure 2.2: Q-Learning algorithm

This method involves a Q-table which contains Q-values for each state-action pair. The table is initialized with zeroes and agent chooses an action according to some approach. The agent performs action and notes the reward obtained and updates the Q-value using Equation 2.9.

$$q(S_t, A_t) \leftarrow q(S_t, A_t) + \alpha[R_{t+1} + \gamma \max_a q(S_{t+1}, a) - q(S_t, A_t)] \tag{2.9}$$

Here $\alpha$ is the learning rate. This process is repeated and after a lot of iterations a good Q-table is ready. Figure 2.2 describes this process.

This method is not feasible for practical RL problems as the Q-table memory consumption rises exponentially as the number of states increase. In next chapter we will discuss some state-of-the-art algorithms which overcome the limitations faced by this approach.

# Chapter 3

# Literature Survey

## 3.1 Related Work

The game of Pursuit-Evasion came in attention for the first time in a book called ”Differential Games” by Rufus Isaacs [2]. Traditional research in the field of Pursuit-Evasion games were based on Game Theory approaches, such as [3], which derive the conditions for capture and optimality and formulate an optimal intercept strategy. Coming to multi agent implementations of the pursuit evasion games, [6] applied this problem to a set of autonomous vehicles trying to catch another vehicle in an unknown environment though coordination and control. Classic open loop control theory approach has also been applied to this task [8]. Most of the optimization and control theory based techniques to formulate the optimal policies of the multiple agents make several unrealistic assumptions such as full system state observability or restriction to move in a connected graph only [7].

Multi-Agent Reinforcement Learning (MARL) has been under active research over the past couple of years due to the rapid advancements in robotics and drone technologies. Instead of using optimizations or control theory, the agents are made to learn their policies through rewards at each step. Some works like [4] have applied reinforcement learning in Pursuit-Evasion tasks. However, these works are based on formulating strategies for pursuers against unintelligent evaders. Reinforcement learning was successfully applied to train an intelligent pursuer against an intelligent evader in [5]. But due to the use of Q-learning for training the agents, their solution cannot be scaled to large state spaces since memory consumption grows exponentially.

Coming to modern state of the art approaches, the use of Deep Reinforcement Learning [9] is being actively used on these multi agent tasks. The challenge is to decide upon the communication and information available to the multiple agents. In some influential work such as [10], the state of the environment as seen by the different agents is modelled as different channels of an image and then the popular Deep Q-Networks [11] is applied to the resulting image to predict discrete actions in a grid. While deep neural networks are applied in traditional DQN, the approaches which model the state as an image generally use CNNs[12] due to the superior performance in detecting features from image data. To enhance cooperation and develop a centralised optimal policy for all the agents, Centralised Learning Decentralised Execution (CLDE) mechanisms are employed. Here the learning is done is done in a centralised fashion by creating one large state action pair corresponding to all agents. However, the execution is decentralised after training, the agents take actions independently. One influential paper that used this developed the Deep Deterministic Policy Gradient (DDPG) approach[13]. Last year, our seniors Chinmay and Naval worked on one of the CLDE algorithms inspired from [14]. But they only implemented a 1 pursuer 1 evader situation because in these algorithms dimension of joint state action space increases proportionally to the number of agents, leading to high computational requirements, along with some unrealistic assumptions.

Communication-based MARL algorithms have been proven effective for large-scale agent cooperation. Earlier works assume that all agents need to communicate with each other. In [15], all agents' hidden states are sent to the shared communication channel, which is used for learning. Communication between all agents leads to high communication complexity and difficulty of useful information extraction.

In Mean Field Reinforcement Learning [16], the multi-agent interactions are approximated as interactions between a single agent and the average effect from some neighboring agents. In this approach, a mean field Q-function (state value function) is defined and it is proven that this converges the standard Q-function. As a result, many agent problem is effectively converted into a two agent problem. Therefore, the computational complexity is reduced significantly since the action value function state space remains constant. There is also cooperation due to shared information between agents.

In the subsequent sections, we explain DQN and Mean Field reinforcement learning in greater detail since they have been employed by us throughout the project.

## 3.2 Deep Q-Networks

Minimising the computation required for calculating Q-functions was under active research since the beginning of the 21st century. In most practical applications, calculating the Q-function at every iteration and representing it in the form of a Q-table is not possible. The number of state action pairs is too large to handle for even modern computers.

Deep Q-Network (DQN) algorithm was formally developed in 2015 by a Google team named DeepMind. This algorithm combined traditional Q-Learning with **deep learning**. Using this algorithm, they solved a range of Atari games with exceptional performance, and published the results in [11].

In DQN, a deep neural network is trained to act as a function approximator to estimate the value of the Q-function. Given a deep neural network with parameters (weights and biases) $\theta$, and a state-action pair $(s, a)$, DQN aims to train the neural network parameters to achieve $Q(s, a; \theta) \approx Q_*(s, a)$. That is, if the state and action corresponding to an agent is passed into the trained neural network, it should output the optimal Q function corresponding to that state-action pair.

The DQN is trained by minimising a sequence of loss function $L_i(_i)$ (that varies with iteration i), given by

$$L_i(\theta_i) = E_{s,a,r,s' \sim \rho(.)}[(y_i - Q(s, a; \theta_i))^2] \tag{3.1}$$

where $s'$ is the next state after taking action $a$ at state $s$, $r$ is the reward obtained, $\rho$ is the distribution over transitions s,a,s',r (behaviour distribution) and $y_i = r + \gamma max_{a'} Q(s', a'; \theta_{i-1})$. Note that $y_i$ is the Q-value estimate using the output from the neural network and the reward obtained in that step.

There are some challenges that affect the stability of training the neural network. In order to overcome these challenges, we look at two 2 techniques known as Target Network and Experience Replay, which are discussed subsequently.

### 3.2.1 Target Network

One major problem is training neural networks for reinforcement learning is that the target is always moving. To train a neural network in a supervised learning setting, we have the input data $x_i$ and corresponding labels $y_i$. The labels remain fixed for a particular input. However, this isn't the case in reinforcement learning. As the agent trains over time, its policy changes and since we do not have the optimal Q function beforehand, we cannot map the input state-action pair (s,a) to a label $Q_*(s, a)$. The neural network tries to minimise the loss function over training iterations. Looking closely at 3.1, we can see that $y_i$ in the loss function which is supposed to be the label is itself not known during training. We are using a function approximator to calculate the optimal Q function at the next state as well. As the neural network trains, the $y_i$ for the same input also changes and this causes instability in the learning and can cause the loss function to diverge.

As a solution, it is proposed to use 2 neural networks in the DQN architecture: the Train network and the Target network. The Train network estimates $Q(s, a; \theta_i)$ in 3.1 and the Target network is used to estimate $y_i$. The weights of the Target network are not updated in every iteration. Instead, after a certain number of iterations, all the weights of the Target network network are replaced by the updated weights of the Train network.

Using a Target network has proven to be extremely successful in reducing the speed at which the target of the neural network varies and increases the stability of training.

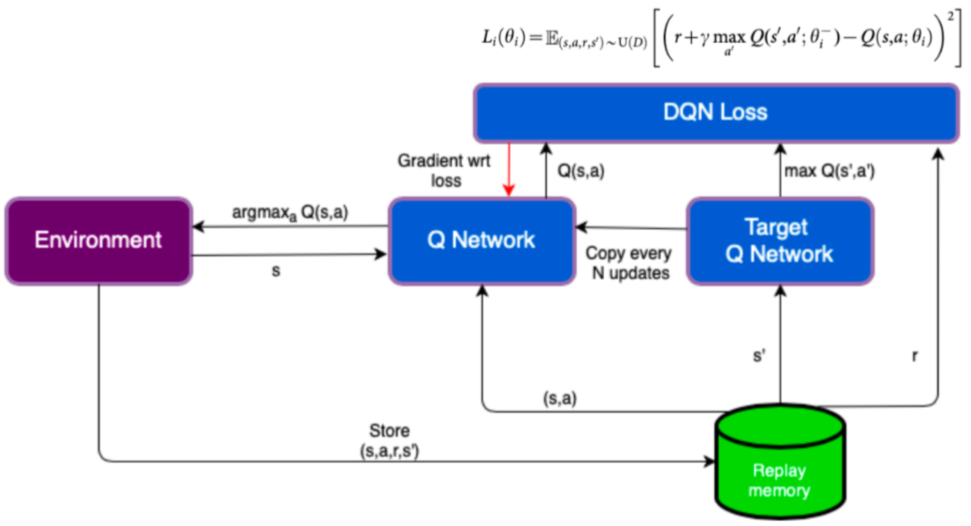


Figure 3.1: Deep Q-Network Schematic

### 3.2.2 Experience Replay

It has been observed through experiments in reinforcement learning that a series of consecutive states are very similar and do not have a lot of new information. This is because an agent in real world is not expected to change its state considerably. Therefore, neural network is trained *online*, i.e if we train the neural network at every step the agent takes during learning phase, then the training inputs to the neural network will be highly correlated. It is well known that neural networks do not train well with correlated training data.

To counter this problem, experience replay was introduced. In experience replay, as the agent takes actions, it stores it observations from the environment in a circular buffer. The experiences are stored in the form of $(s, a, s', r)$ where the variables corresponding the states, action and reward corresponding to one step. During training, experiences are sampled randomly from the buffer in the form of mini-batches and then this is used in one training iteration of the neural network. Typical mini-batch sizes are 16, 32, 64 and 128.

This has two advantages:

- Since we are sampling randomly from the buffer, we are using diverse and uncorrelated data to train the neural network which leads to better accuracy and faster training.
- Since we are passing minibatches as training data and not just a single input at each training iteration, we can make use of the vectorized implementations in machine learning which saves a lot of computation and time.

## 3.3 $\epsilon$-Greedy Approach

Given action-values function for each state-action pair, at each time step there is an action with the greatest value. If we select the action with best Q-value then we are **Exploiting** our current knowledge of the values of actions and being greedy. On the other hand, if we choose a non-greedy action then, we our **Exploring** the environment and improving our knowledge. Exploitation leads to high expected reward in a single time step but exploration may lead to higher rewards in long run. Exploration results in lower rewards in short run but leads to higher rewards eventually because by taking random actions it reaches those states which it might not have reached by just exploitation.

This tradeoff between Exploration and Exploitation is handled using $\epsilon$-Greedy approach. Using this approach we choose a random action with probability $\epsilon$ or choose an action with best Q-value with probability $1 - \epsilon$. This way by choosing appropriate value of $\epsilon$, we can balance Exploration and Exploitation. The value of $\epsilon$ is kept high during initial phase of training because the current knowledge is very poor to exploit. As the training progresses, the value of $\epsilon$ is decreased so that we start exploiting knowledge as it keeps getting better. Equation given below shows this approach mathematically.

$$a_t = \begin{cases} \text{argmax}_a Q(s,a) & \text{with probability } \epsilon \\ \text{randomly chosen action} & \text{with probability } 1-\epsilon \end{cases}$$

## 3.4 Mean Field Reinforcement Learning

In practical multi-agent reinforcement learning problems, standard approaches often fail due to the curse of dimensionality as the state space grows exponentially with number of agents. Mean Field theory can be applied to multi-agent RL problems in order to improve scaling of the system to large number of agents [16] [17]. The main idea in this approach is that interaction between multiple agents can be simplified as interaction between a single agent and the average effect of the surrounding population. Figure 3.2 shows this interaction between an agent and the average effect of local population (blue area). In this way, multi-agent RL problems boil down to single agent problems and thus, allowing high scalability.

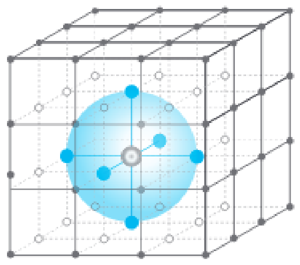


Figure 3.2: Interaction between single agent and average effect of neighbours

To model this interaction between single agent and average effect of neighbours, the joint Q-function for agent with index $j$ is simplified as shown in Equation 3.2. In this way Q-function is factorized using pairwise local interactions between agents.

$$Q^j(s, \mathbf{a}) = \frac{1}{N^j} \sum_{k \in \mathcal{N}(j)} Q^j(s, a^j, a^k) \tag{3.2}$$

Here $\mathcal{N}(j)$ is the set of neighbours of agent with index $j$ which can be decided based the level of observability required in different scenarios. Breaking down of Q-function in this fashion does not affect the global interaction between agents.

When action space is Discrete, action $a^j$ can be represented in one-hot encoding format, on the basis of action chosen out of the available action choices. Thus, each component of $a^j$ vector signifies if that action was chosen or not. Action taken by agent with index $j$ is written as: $a^j = [a^j_1, a^j_2......, a^j_D]$ for $D$ possible action choices. The *mean action* $a^{-j}$ can be written as shown in Equation 3.4 where each neighbour's action is denoted by $a^k$ (one-hot encoded format). So, $a^k$ for each neighbour can be represented as a sum of the *mean action* $a^{-j}$ and small fluctuation $\delta a^{j,k}$ as shown in Equation 3.3

$$a^k = a^{-j} + \delta a^{j,k} \tag{3.3}$$

$$a^{-j} = \frac{1}{N^j} \sum_k a^k \tag{3.4}$$

As shown in Equation 3.2, we can break down the joint Q-function as interactions between individual agents. Applying Taylor's Theorem, the pairwise Q-function can be expanded as shown in Equation 3.5

$$\begin{aligned}
Q^j(s, \mathbf{a}) &= \frac{1}{N^j} \sum_k Q^j(s, a^j, a^k) \\
&= \frac{1}{N^j} \sum_k [Q^j(s, a^j, a^{-j}) + \nabla_{a^{-j}} Q^j(s, a^j, a^{-j}).\delta a^{j,k} \\
&\quad + \frac{1}{2}\delta a^{j,k}.\nabla^2_{a^{-j,k}} Q^j(s, a^j, a^{-j,k}).\delta a^{j,k}] \\
&= Q^j(s, a^j, a^{-j}) + \nabla_{a^{-j}} Q^j(s, a^j, a^{-j}). \left[\frac{1}{N^j} \sum_k \delta a^{j,k}\right] \\
&\quad + \frac{1}{2N^j} \sum_k \left[\delta a^{j,k}.\nabla^2_{a^{-j,k}} Q^j(s, a^j, a^{-j,k}).\delta a^{j,k}\right] \\
&= Q^j(s, a^j, a^{-j}) + \frac{1}{2N^j} \sum_k R^j_{s,a^j}(a^k) \approx Q^j(s, a^j, a^{-j})
\end{aligned} \tag{3.5}$$

Here $R^j_{s,a^j}(a^k)$ denotes the Taylor polynomial's remainder. Thus, the joint Q-function is first broken down into pairwise interaction Q-function, which is then broken down into interaction between agent $j$ and a *virtual mean agent* whose action is represented by $a^{-j}$. $a^{-j}$ is characterized by neighbour actions undertaken following previous iteration's policy. This mean Q-function can be approximated using Neural Networks in a similar fashion as in the case of Deep Q-Learning discussed in Chapter 3.

This approach is highly scalable and combining it with Deep Q-Learning approach can lead to effective training of a large number of agents in practical RL problems.

# Chapter 4

# Problem Formulation and Solving Approach

## 4.1 Game Setting

- Pursuers and evaders are agents in a discrete $20 * 20$ grid
- At each time step, all the agents take an action and may move to a neighbouring cell of the grid
- Aim of the pursuers is to catch the evader as soon as possible
- Aim of evader to is resist capture by the pursuers and stay far from it
- None of the agents can go outside the boundary wall
- The game continues until evader is caught or set time has elapsed
- Pursuer wins if it is able to catch the evader within a fixed time otherwise the evader wins

## 4.2 Pursuit Evasion game modelled as a reinforcement learning problem

In reinforcement learning, the final objective of each agent is to maximise the rewards obtained and minimise the costs. To formulate any problem as an RL problem, we need to set a goal for the agents and reward them throughout the duration of an episode.

The pursuit evasion problem can easily be modelled as a reinforcement learning problem, The pursuers and evaders clearly have well defined goals. Also, since the performance of the agents depends on their relative distance in the grid, they can be rewarded based on that. There is also a clear definition of when an episode ends; either the evader is caught, a certain number of time steps occur, or any agents crosses the boundary wall of the grid.

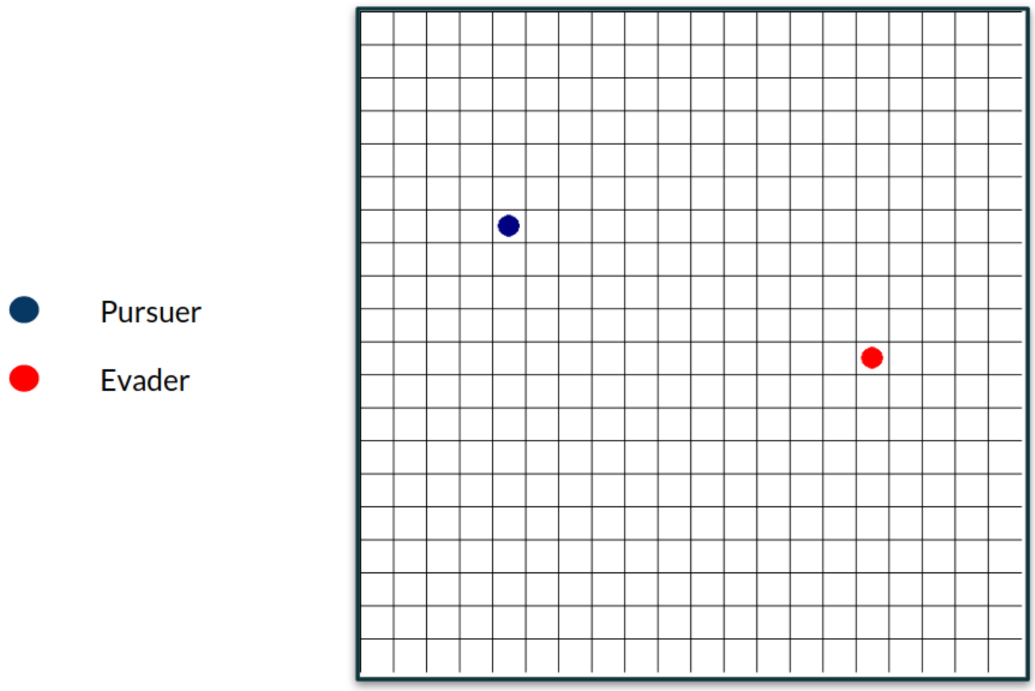


Figure 4.1: Game Setting with 1 Pursuer and 1 Evader

In our game setting, the state space is finite. There are $20 * 20 = 400$ possible positions and so that is the size of the state space for a single agent. The combined state space of all the agents obviously depends on the number of agents. The action space is also finite since the agents can choose their actions at any time from a certain set of actions which is finite(in our implementation, evader has twice the number of possible actions as the pursuers).

The learning is done *offline*: The agents takes actions and stores experiences into the replay buffer and the learning based on these experiences happens later when that experience is sampled.

We have implemented and compared three different algorithms. They vary in the *observability* of the agents, which is the information that the agents have about the other agents.

## 4.3 Reward Structure

The choice of reward function in reinforcement learning depends on the problem and is crucial to the learning process. An ideal reward function should handle the various situations possible during the episode and should be rich in instructive information. We discuss two reward structures below

### 4.3.1 Naive Reward Strategy

- Pursuers are rewarded -1 for each time step and evader is rewarded -1
- An agent is rewarded +50 if the episode ends in its favour and thus the other agent get -50 as a reward.
- If any agent crosses the boundary of the wall, it is rewarded -50.

This reward strategy doesn't work very well. Although the reward structure at each time step encourages the evader to not get caught and the pursuer to catch the evader, the agents are not making use of their observability and their relative positions. The evader isn't learning to go far from the pursuers and likewise the pursuers don't learn to go near the evader. The huge negative reward on crossing the wall works well though.

### 4.3.2 Optimal Reward Strategy

- At every time step, each pursuer is awarded a negative reward proportional to the *Euclidean distance* between that pursuer and the evader
- At every time step, the evader is awarded a positive reward proportional to the *Euclidean distance* between it and the **nearest pursuer**
- The proportionality constant normalises the reward magnitude to make it less than 1
- If any pursuer catches the evader, it gets a +10 reward and the evader consequently gets a $-10$ reward.
- If any agent goes crosses the walls of the grid, it receives a $-5$ reward.

The final reward structure for a pursuer $p_i$ and the evader e chosen by us can be summarized as:

$$Reward(p_i) = \begin{cases} -\text{Euclidean Distance}(p_i, e)/k & \text{every time step} \\ +10 & \text{catches evader} \\ -5 & \text{crosses grid boundary} \end{cases}$$

$$Reward(e) = \begin{cases} +\text{Euclidean Distance}(p_i, e)/k & \text{every time step} \\ -10 & \text{gets caught} \\ -5 & \text{crosses grid boundary} \end{cases}$$

where $p_{i*}$ is the pursuer closest to the evader and k (proportionality constant) is kept as $20\sqrt{2}$ because the grid size is 20.

This reward structure works really well since the observations that the agents make during learning further helps them decide the optimum policy.

## 4.4 Implementation of three approaches

For the course of this project, we implemented and compared three different algorithms to solve the pursuit evasion problem. The algorithms differed between their complexities, agents' observabilities and neural network inputs.

### 4.4.1 Independent Agents Approach

- In this implementation, all the pursuers are assumed to be independent. None of the pursuers are expected to know the positions of other pursuers.
- Each pursuer takes develops it policy and takes actions just on the basis of its own position and the evaders position. This is the case of *restricted observability.*
- Since each agent has just 4 positional coordinates to take actions, it is expected that this algorithm scales well. Even if the number of agents increases, the size of input to the Train and Target network remains the same.
- Due to the independent policies, this algorithm isn't expected to have a very good performance since there is no cooperation between the agents to develop a mutual winning policy.

### 4.4.2 Full State Space Approach

- In this implementation, all the pursuers know the position of the other pursuers.

- Each pursuer takes into account the positions evader and other pursuers in developing its policy and taking optimal actions. This is the case of *full observability.* There is one combined state space of all the agents.
- The number of inputs to the Train and Target network scales with the number of agents. Therefore, if the number of agents is increased to a large number(greater than 5-7), then the number of features are so large that it would require a very huge amount of training data to minimise the loss function. This comes under the *curse of dimensionality*, often encountered in machine learning applications. As a result, this algorithm is not expected to scale very well with the number of agents
- Theoretically, if we provide enough training data and computation resources, this approach is expected to perform the best since an optimal policy can be learned considering the relative positions of all the pursuers and the evader for any step. There is inherent cooperation between the pursuers.

### 4.4.3 Mean Field Approach

- In this implementation, all the pursuers know the actions (**not positions**) of the other pursuers which are within a certain radius of that pursuer. This is the case of *partial observability.*
- Each pursuer uses these actions of some the other pursuers to create a an effective Mean Field agent according to the Mean Field Theory in the previous chapter.
- Therefore, even if the number of agents is increased, the problem effectively reduces to a 2 pursuers 1 evader problem from the point of view of a single pursuer.
- The number of inputs to the Train and Target network does not scale with the number of agents. This is because we are passing the position of the pursuer, position of the evader and the mean of the actions of the neighbouring pursuers into the neural network. Therefore, the input size always remains constant irrespective of the number of agents. This allows a feasible computation overhead and training data required.
- Theoretically, the Mean Field Q-function should converge to the optimal Q-function for each pursuer. But since we are estimating Q-values using deep learning, there is bound to be a saturation loss value. However, the partial information from the nearby pursuers can help the pursuer in deciding its future actions. For example:

If all pursuers in the neighbourhood of a pursuer are going towards right direction to catch the evader, then this extra information can help the pursuer in deciding to go right itself.

- This approach has the key benefits of both of the other two approaches; namely the complexity and scaling benefits along with the performance benefits due to cooperation.

## 4.5 Deep Q-Networks Implementation

We have implemented a *distributed DQN* model where each agent has its own Train and Target networks.

1. Input vector is the observation received by the agent. This observation varies for each of the three algorithms compared. Also, the input can be purely positional coordinates or can also contain mean of actions (the extra input in mean field implementation). Figure 4.2 details the size of the input vector for the three approaches.

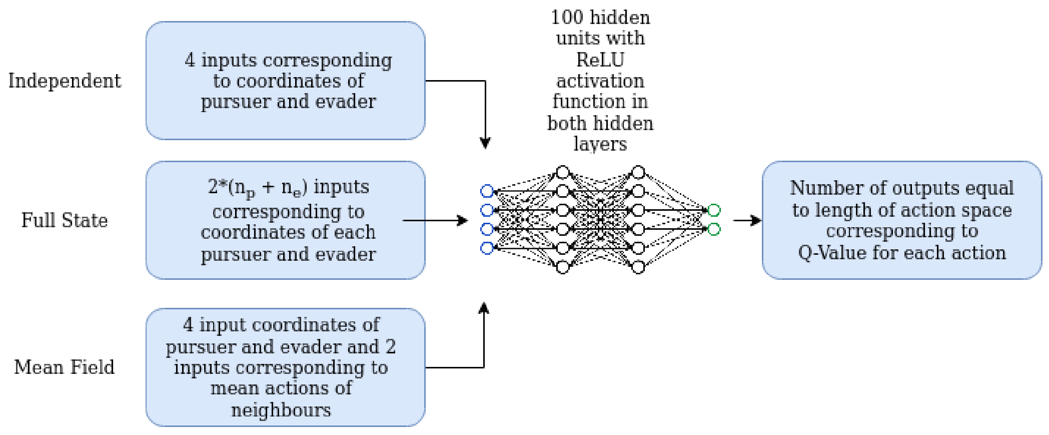

Independent
4 inputs corresponding to coordinates of pursuer and evader
Full State
2*($n_p$ + $n_e$) inputs corresponding to coordinates of each pursuer and evader
Mean Field
4 input coordinates of pursuer and evader and 2 inputs corresponding to mean actions of neighbours
100 hidden units with ReLU activation function in both hidden layers
Number of outputs equal to length of action space corresponding to Q-Value for each action

Figure 4.2: DQN Implementation for three approaches

2. The neural network is a fully connected network. It has two hidden layers with 100 neurons each. The size of the network was chosen on the basis of experimentation to optimise the performance by introducing more learnable weights, all within reasonable computational requirement. The activation function used in the input and hidden layers is **ReLU**(Rectified Linear Unit). This is chosen since it outperforms other common activation functions like sigmoid and *tanh* in training performance.

3. The output layer has a size equal to the number of possible actions for the agent. Each of the output neuron gives the *action-value function* corresponding to one of the actions. We have not used an activation function at the output layer. Therefore, the output directly gives the expected future reward of taking a particular action from that state. The agent takes actions corresponding to the highest activated neurons in the output layer during exploitation phase of $\epsilon - greedy$ strategy. Therefore, our policy is *deterministic* i.e we take the best possible action at each step.

4. **Note:** Had we taken **softmax** activation in the final layer, we could have had a *stochastic* policy, where each action is taken with a probability according to the action-value function. We did not choose this approach since we want our pursuers to take the best actions and maximise their performance of catching the evader in minimum time steps.

# Chapter 5

# Implementation and Code

Our implementation consists of three python notebooks, one for each approach that we discuss. Each python notebook consists of different classes for setting up the environment, facilitating training and final testing and plotting. In this chapter we discuss all this in detail along with the dependencies and libraries used.

Our work deals with applying three different training approaches to agents and study their performance in different settings. As explained in Section 4.4, we will apply these three approaches to train multiple pursuers against an evader. As pursuers would be numerous, the evader will be given additional ability (twice speed) so that pursuers have to come up with some sort of cooperation to be successful. Each agent will have its own Train Network, Target Network and Experience Buffers. We would be using Deep Q-Networks for training the agents. The pursuers' training will be done using three different approaches as explained in previous chapter. The codebase is almost same for the three approaches except that the neural networks would take in different inputs.

## 5.1 Dependencies

1. **OpenAI Gym:** OpenAI Gym is a standard toolkit to compare and test algorithms in Reinforcement Learning. It enables us to set the environment for the game.

2. **Tensorflow:** We used Tensorflow 2.0 for the training of Deep Q-Networks. It gives easy compatibility with GPU for faster training.

3. **Pygame:** Pygame is an open-source library for the development of games using python. We use Pygame 2.0.1 for rendering our game and visualizing the policies learnt by agents. It dynamically visualizes the steps of agents and provides functionality to create GIFs of the gameplay.

## 5.2 Pseudo-Code

**Algorithm 1:** Learning the policy for agents

```
Take in value of numPursuers and numEvaders;
Initialise Pursuer and Evader Agent() objects and the environment;
Initialise epsilon;
for episode in num_training episodes do
    Randomly initialise the position of evaders and pursuers;
    step_count =0;
    while episode does not terminate do
        Old State = $s$;
        Create a neural network input vector from state information;
        Pass input into TrainNet;
        if random number $\leq \epsilon$ then
            Take random action;
        else
            Take action $a$ corresponding to max of output layer neurons;
        end
        New state = $s'$, Reward obtained = $r$;
        if step_count == 200 OR any pursuer catches evader then
            terminate episode;
        end
        Store (s,a,s',r) into experience reply buffer;
        Pass information about $s'$ into TargetNet to estimate the value of
          $max_{a'}Q(s', a')$ ;
        Compute $y_i = r + \gamma max_{a'}Q(s', a'; \theta_{i-1})$;
        Select minibatch of samples;
        Train the TrainNet using its output and $y_i$ values by optimising loss
          function in 3.1;
        if number_training_iterations == $C$ then
            Copy weights from TrainNet to TargetNet;
        end
    end
end
```

## 5.3 MAgent Framework

After creating the basic model, we ported all our implemented code into the MAgent framework format [18]. Unlike other research platforms are compatible with a single or a few agents, MAgent provides the functionality of creating 2D grid like environment for reinforcement learning applications with upto a *million* agents. We choose MAgent because unlike the previous work done at IIT Delhi for pursuit evasion games on just 2 agents, we wanted to see the effect of scaling the number of agents. The Magent framework has certain rules for creating the classes in a hierarchical format. Porting our code to this format made our code structured and on a level with the state of the art research done in this field.

## 5.4 Code

Our code is divided into three Python notebooks, one for each training approach that we discuss namely: Full State, Independent and Mean Field Approach. The codebase for all three notebooks is similar except some functions which are required for training purposes. All the classes and important functions of the codebase are explained in this section. View code on Github

**Class: DQN()**

This class is used for initializing and training the Deep Q-Networks. It initializes the experience buffer and the optimizers for Neural Network It assigns random weights to Neural Networks and initializes it with given layers, number of neurons and activation functions. Some important functions of this class are explained below.

1. *train():* Samples data into given batch size and backpropogates the error obtained using TargetNet and trains TrainNet using Adam Optimizer.
2. *get_action():* Uses Epsilon Greedy Approach to either select random action or feed-forward TrainNet to obtain action at a given state.
3. *add_experience():* Adds the current experience to the experience buffer. If buffer size exceeds the maximum buffer size allowed, then it pops the oldest experience available.

4. *copy_weights():* Used to copy the weights of Train Network into Target Network after fixed number of iterations.

**Class: Agent()**

This is the base class for agents in our environment. Some important elements in this class are:

1. *position:* Gives the coordinates of agent's location in the grid world
2. *isPursuer:* Mentions if the agent is a Pursuer or an Evader
3. *rewards:* List with cumulative rewards of agent in each episode
4. *losses:* List with average loss of agent's Neural Net in each episode
5. *TrainNet:* The Training Network of agent
6. *TargetNet:* The Target Network of agent

**Class: World()**

This class is used to set up the agents in the environment. It sets up custom number of pursuers and evaders and initializes them at different locations of the grid. It sets up the action space for pursuers and evaders as required by the game setting. Some important functions of this class have been discussed below.

1. *initialize_pursuers():* This function initializes the pursuers at random locations in the grid. Custom locations could also be given.
2. *initialize_evaders():* This function sets up the evaders at either random locations or custom locations of the grid as required by the game.
3. *reset_world():* It resets all the pursuers and evaders back to initial positions. This is required when an episode ends.
4. *get_NN_input():* This function converts the state into required input for the neural network according to the 3 training approaches that we deal with (Full State, Independent and Mean Field). This is the function which is different across the three notebooks.

5. *initialize_NN():* It initializes the neural networks for the agents according to their status (pursuer or evader). All important hyperparameters for training are passed in this function. It can either load previously saved models or initialize new model with the given hyperparameters.

**Class: MarlEnv()**

This is the class which defines our environment according to the standard rules of a Gym environment. The functions of this class are explained below.

1. *__init__():* It first populates the world with appropriate number of pursuers and evaders in required initial positions.
2. *reset():* To reset the world after an episode ends. It re-initializes the agents in the environment.
3. *step():* Given a state and actions for agents, it applies the actions to respective agents in the environment. It computes and returns the reward obtained by agents after taking the given action and returns the next state along with it. It also checks if the episode has terminated and returns the status of episode termination.
4. *render():* After each step this function updates the Pygame display in order to visualize the agents and their actions. It displays the full game and has the functionality to save the gameplay as GIF.

**Function: play_game()**

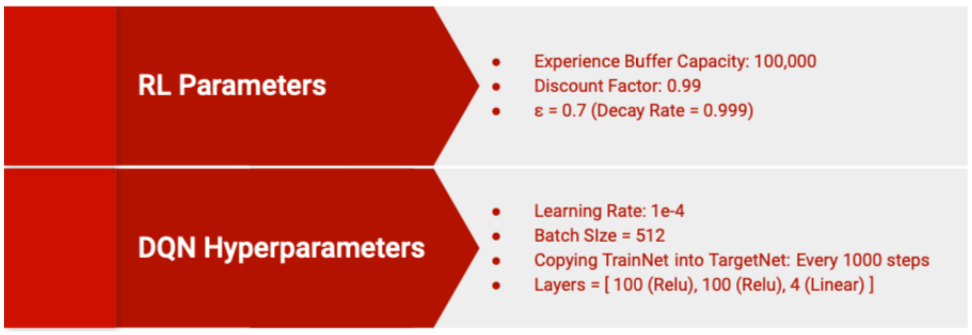
This function is used to play an episode of the game. It takes in *MarlEnv()* as input and returns the respective rewards earned by agents, losses of their neural nets and the result of the game. It copies the weights from Target Network to Train Network after a certain period of iterations and maintains the experience buffers for agents. All the training information gets stored for plotting purposes.

# Chapter 6

# Experiments and Results

## 6.1 Selecting Parameters

To carry out the experiments, it was important to set the parameters to be used in the reinforcement learning and deep learning algorithms. Since we implemented diverse algorithms such as $\epsilon - greedy$, DQN and Target Network, there were several possible combinations of parameters to choose from. Through various experiments and literature survey throughout the course of the project, we finalised upon the parameters listed in Figure 6.1.



**RL Parameters**

- Experience Buffer Capacity: 100,000
- Discount Factor: 0.99
- ε = 0.7 (Decay Rate = 0.999)

**DQN Hyperparameters**

- Learning Rate: 1e-4
- Batch SIze = 512
- Copying TrainNet into TargetNet: Every 1000 steps
- Layers = [ 100 (Relu), 100 (Relu), 4 (Linear) ]

Figure 6.1: Selection of parameters for the experiments

## 6.2 Base Experiment: 1 Pursuer vs 1 Evader

### Problem Setting

Both pursuer and evader are smart and dynamic and have their own Train Network and Target Network to determine their respective Q-functions. Evader starts randomly and moves in 4 possible directions trying to evade the pursuer. Pursuer starts randomly in the grid tries to catch the moving evader, by moving in 8 possible directions.

### Training

This model was trained for 1000 episodes. Figure 6.2 shows the rewards earned and losses during training for pursuers. Figure 6.3 shows the plots for evader rewards and losses during training. 1 epoch consists of 10 episodes.

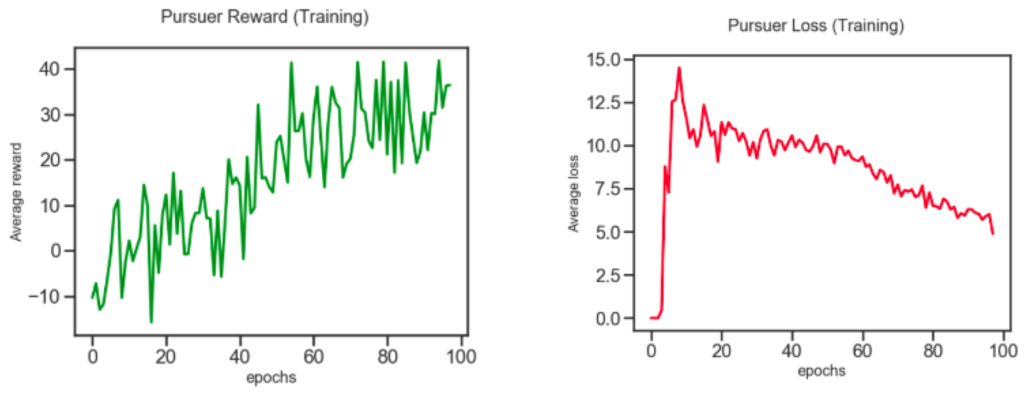


Figure 6.2: Pursuer rewards and losses

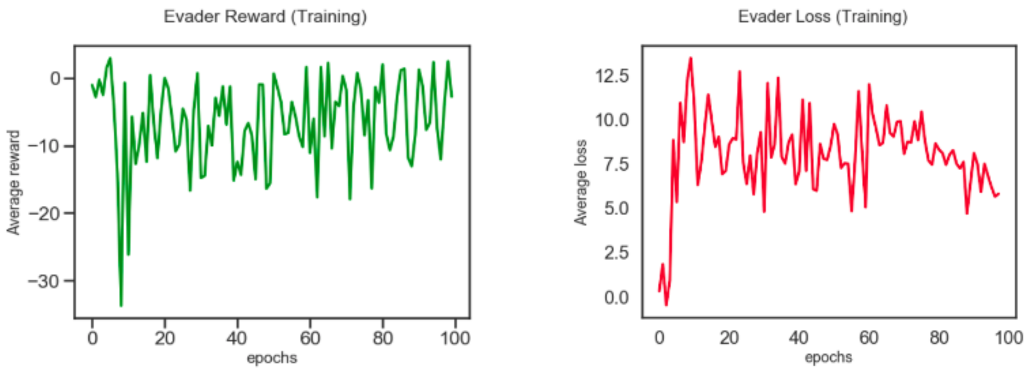


Figure 6.3: Evader rewards and losses

The initial reward for pursuer is negative as the agent is trying to learn the walls of grid and keeps on getting a reward of $-5$ whenever it crosses the boundary. As we can see the pursuer reward increased signifying that the pursuer became smart enough to catch the evader and learn not to go outside the grid and the training loss decreased eventually. The evader loss also decreases eventually but the reward earned by evader stays almost the same. This is because of the difference in ability of pursuer and evader. After training evader was not able to be successful, since pursuer had more degrees of freedom, and thus, evader received negative reward throughout.

### Testing

Testing is just like training but with $\epsilon = 0$. That is, all the agents take the best possible action at each step(exploitation). On testing on 500 episodes, the pursuer was able to catch the evader most of the times ( 490). This is perhaps due to difference in ability between the pursuer and evader. The pursuer really utilises its ability to move diagonally to intercept the evader.

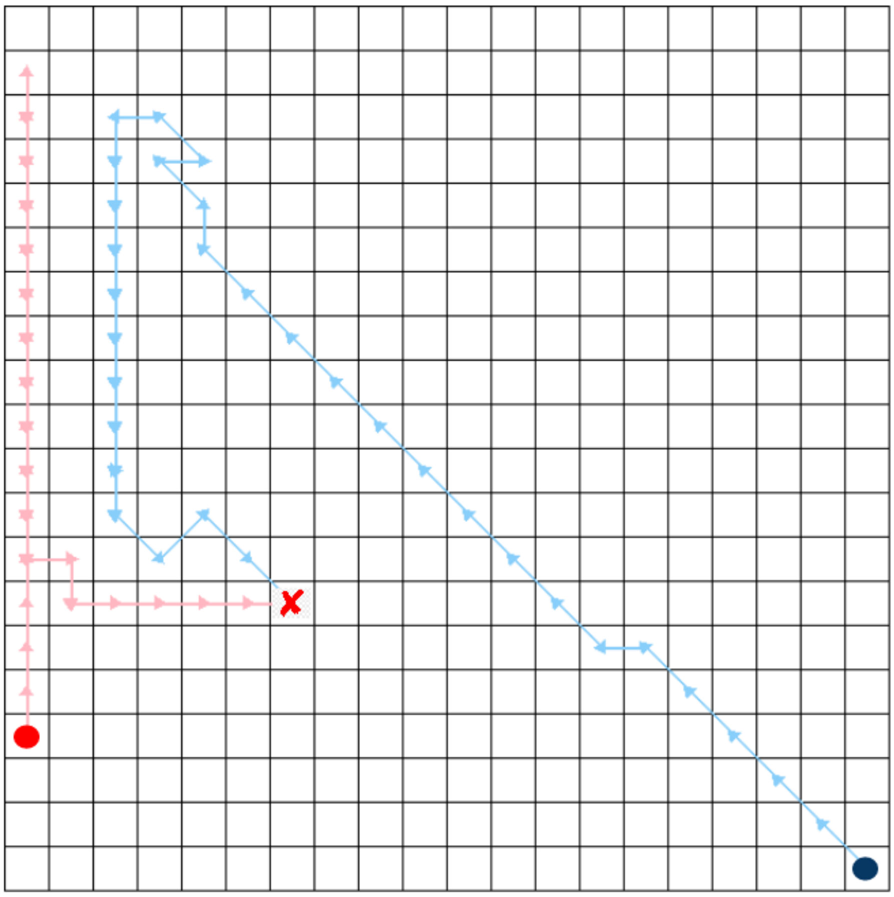


Figure 6.4: Path Trace [Blue: Pursuer and Red: Evader]

1. The pursuer and evader start from randomly initialised positions, marked by the blue and red dots. Initially, The pursuer is at the bottom-right on the evader is at the bottom-left,

2. The pursuer sees the evader moving up and so instead of just moving left to reach the evader's position, it moves diagonally upwards in order to intercept the evader.

3. As the pursuer gets close to the evader, the smart evader quickly changes it's direction completely and starts going down. The pursuer retaliates by itself changing its direction.

4. Since the evader has learned not to cross the wall, it moves right in order to try to increase its distance from the pursuer and get more positive reward. But the pursuer moves diagonally and finally catches the evader.

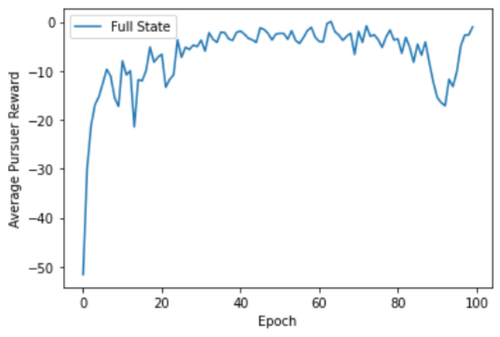
## 6.3 Experiment: Multiple Pursuers vs 1 Evader

This experiment aims to compare the performance of 4 pursuers against 1 evader using three different approaches for pursuer training. Pursuers can move in 4 directions whereas evaders can move in 8 directions (including diagonals) with twice the speed of pursuer. As the evader has more capability compared to pursuers, the pursuers will have to form cooperative strategies in order to succeed. For each approach, the evaders have the same training approach (Full State) for consistency, in order to compare pursuer performance. This experiment has three parts, one for each kind of training approach for pursuers and at last the comparison between these approaches is discussed in detail.

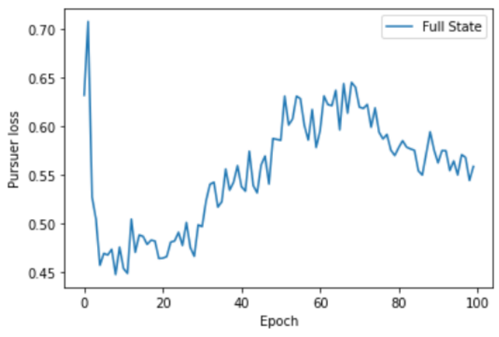
#### Training

1000 episodes were run for each training approach. 1 epoch consists of 10 episodes. Figure 6.5 shows the training plots for average rewards earned by pursuers and losses faced. Whenever an agent goes outside the grid, we call it a violation. Figure 6.6 shows the number of violations for pursuers as training progresses.

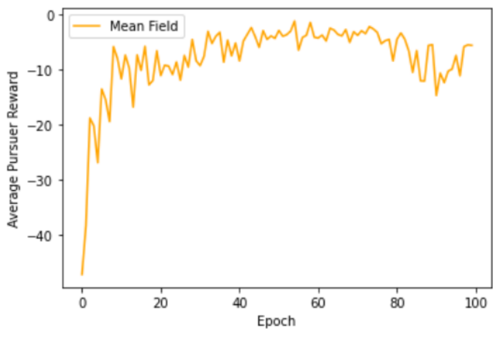
We can observe that the rewards obtained start from highly negative values and saturate at the end for all three approaches. Highly negative rewards in initial phase of training is attributed to increased violations of pursuers as they haven't learned the boundaries of the grid. Figure 6.6 shows that violations were really large initially for all three approaches. As training progresses, the number of violations decrease signifying that pursuers have learnt the boundaries successfully. Final phase of training sees saturated rewards of the pursuers as they are able to catch the pursuer almost every time. The losses also decrease eventually as the Train Network is able to learn the optimal Q-values through utilizing the experience buffer.



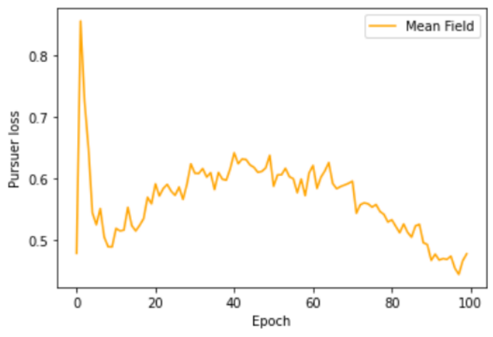
(a)



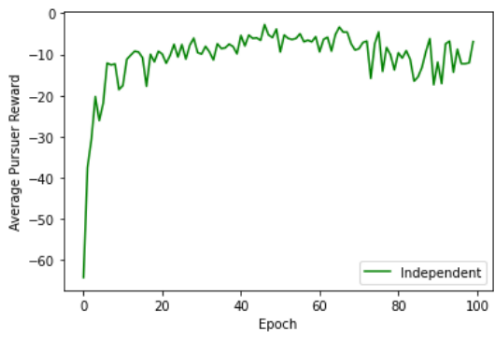
(b)



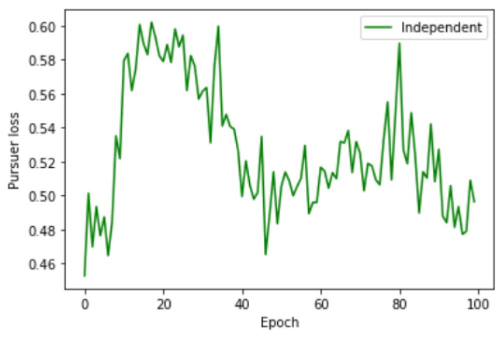
(c)



(d)



(e)



(f)

Figure 6.5: Training Rewards and Losses for Pursuers

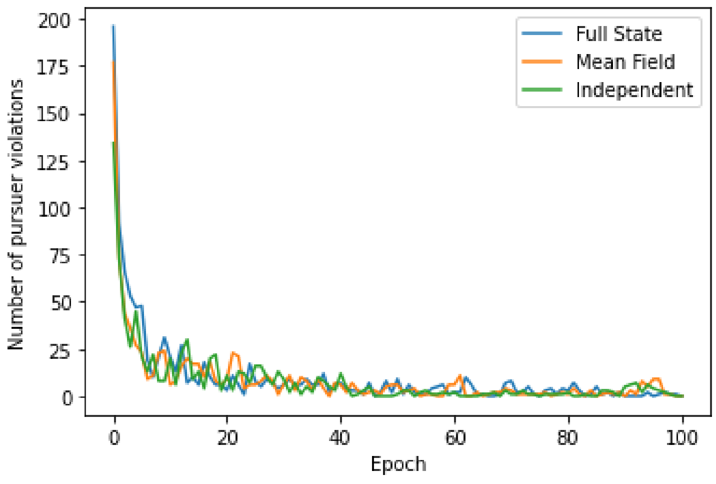


Figure 6.6: Pursuer Violations

### Testing

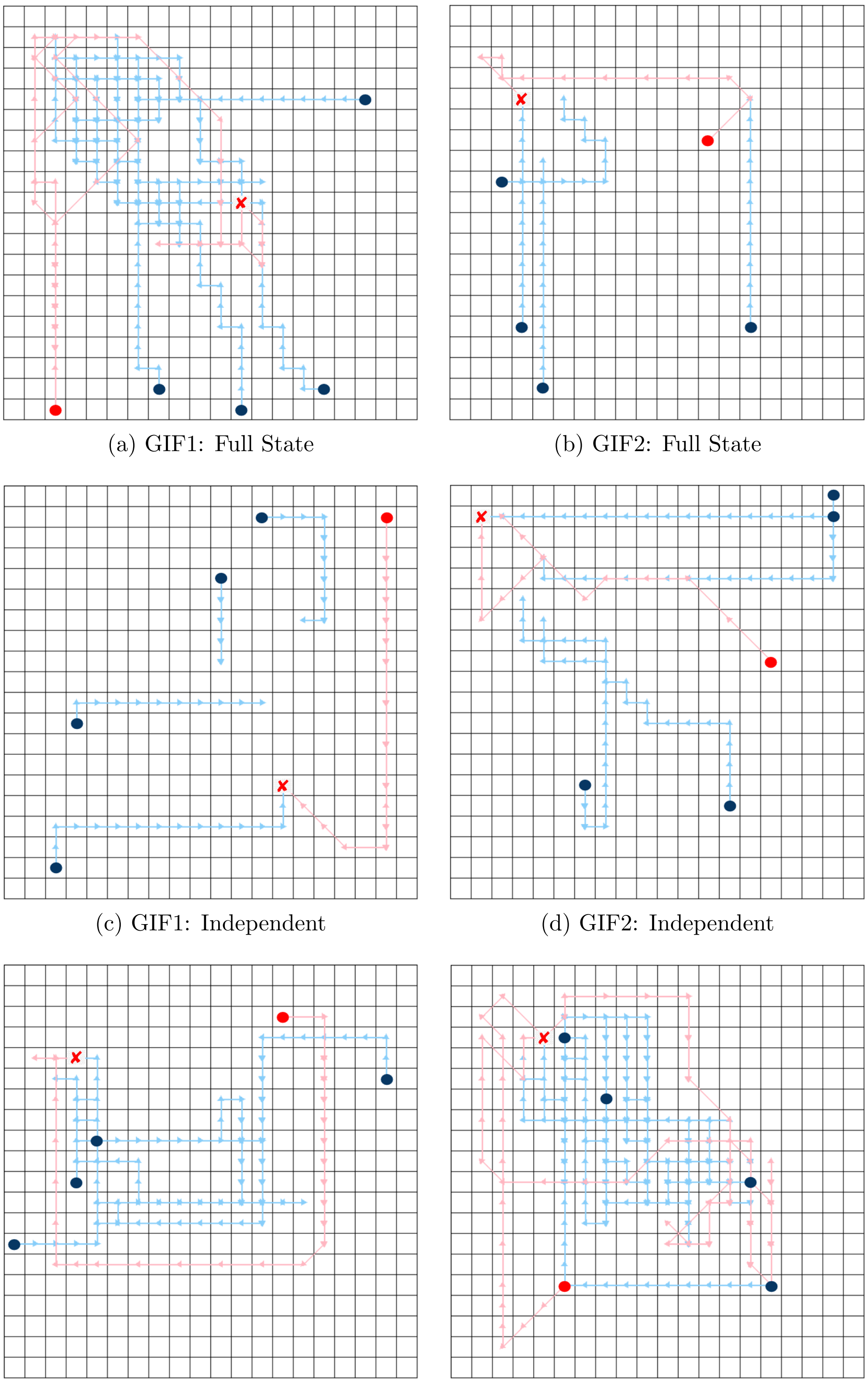
After training for 1000 episodes with each approach, we test the performance of agents. Figure 6.7 shows path traces for some episodes during testing with the three approaches for pursuer training.

1. Figure 6.7 contains the snapshots of a few of the simulations of the testing carried out by us using the PyGame python framework. The complete GIFs can be viewed here: View GIFs on Github

2. It is clearly evident from the GIFs that in most cases, the pursuers are ultimately able to catch the evader. In few cases (mostly from independent agents approach), the pursuers perform very poorly and are unable to develop a strategy to catch the evader.

3. In some cases, the evader uses its ability to move 2 steps at a time to quickly escape the approaching pursuers and go to the other end of the grid.

4. An interesting observation is that a lot of times, the pursuers choose to stay at their position if they feel that the evader will move towards them while running from the other pursuers. Also interesting is how all the agents suddenly change their directions when approaching the walls of the grid. They have learned this due to the high negative reward.


(a) GIF1: Full State (b) GIF2: Full State

(c) GIF1: Independent (d) GIF2: Independent

(e) GIF1: Mean Field (f) GIF2: Mean Field

Figure 6.7: Snapshots of simulation View GIFs on Github

## 6.4 Comparison of performance between the three approaches

### 6.4.1 Training performance

For the same selection of parameters and training on the same number of episodes, the pursuer rewards for training in all the three approaches go from a highly negative value to near zero. There is not much difference between the curves, except that the full state space approach is slightly higher than mean field approach followed by independent agents approach. This was expected since as we had discussed before, full state space approach provides more information about the environment to all the agents and they may cooperate to yield higher rewards.

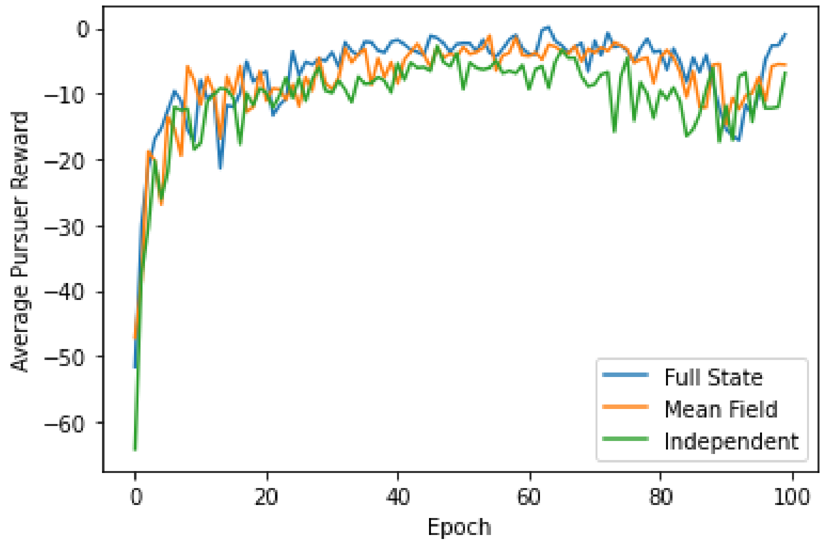


Figure 6.8: Comparison between average pursuer training rewards (1 epoch = 10 episodes)

The DQN loss curves for all the three approaches are quite noisy and erratic. This is because the evader is also learning along with the pursuers. The evader also gets smarter with time and so the optimal actions predicted by the TrainNet may change over time. This causes the high noise in the plots. An interesting thing to note is that in the mean field approach, the loss decreases rapidly after about 50 epochs (500 episodes).

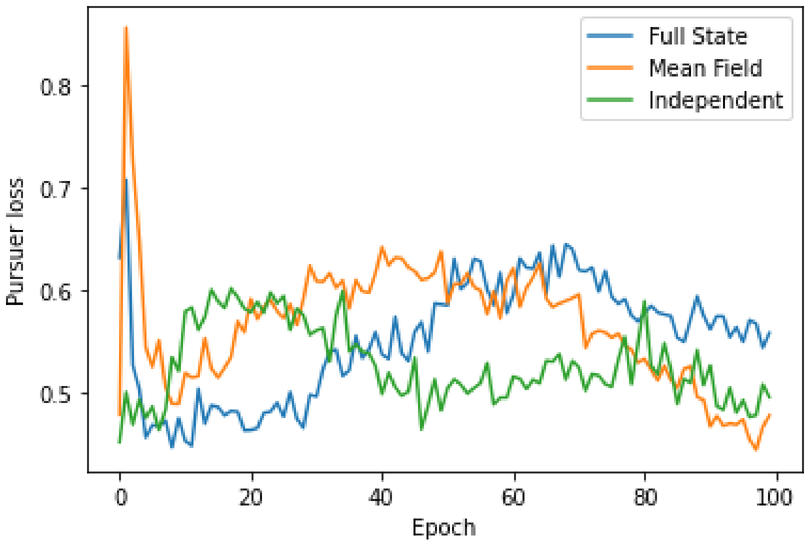


Figure 6.9: Comparison between average pursuer DQN loss (1 epoch = 10 episodes)

### 6.4.2 Communication between agents

In the independent agents approach, since all agents act independently, there is no exchange of positions or actions between the agents. However, there is an exchange of nearby agents' actions in mean field approach, and all agents' positions in full state space approach. This leads to a lot of messages being exchanged between the agents.

In real world applications, lower number of messages between agents is desirable since more messages can choke the network channel between the agents and leads to more loss of messages. Also, sending and receiving messages requires additional power and computation in real life agents.

| **Approach** | **Number of messages exchanged (1000 episodes, 4 agents)** |
|---|---|
| Independent | 0 |
| Mean Field | 283032 |
| Full State Space | 534032 |

Table 6.1: Total messages exchanges between the four pursuers (Training)

6.1 shows the number of messages exchanged between the pursuers while training them for 1000 episodes under the three different approaches. The radius of communication between any two agents in mean field approach was set to 10.

Looking at the results, the mean field approach has **47% less number of messages** as compared to full state space approach. Alternatively, the full state space approach has 88% more messages than the mean field approach. This result is very important in our final conclusion between the approaches.

### 6.4.3 Absolute performance on testing

After training, all the three approaches were tested with $\epsilon = 0$ (exploitation) on 500 episodes. A successful episode is one where any of the pursuer is able to catch the evader before the episode terminates.

| **Approach** | **Number of Successful Episodes** |
|---|---|
| Independent | 226/500 |
| Full State Space | 482/500 |
| Mean Field | 465/500 |

Table 6.2: Number of successful episodes during testing

Table 6.2 shows the number of successful episodes averaged over the four runs performed by us. The results are extremely interesting. As expected, the full state space approach has the best performance followed by the mean field approach and finally the independent agents approach.

The striking observation is that just exchanging actions between nearby agents, as done in the mean field approach, increases performance to such an extent that mean field approach has a **113% improvement** over the independent agents approach in absolute performance. Compare this to the meagre 3.6% improvement obtained by the full state space approach over the mean field approach even though all the state information is shared.

### 6.4.4 Time taken to catch the evader

Next, we looked at the actual time steps taken by the pursuers to catch the evader. This was done to check if there are lots of successful episodes but the agents take a lot of time to catch the evader. The results are summarized in the form of a probability of catching the evader in a given number of allowed timesteps.

Ideally, a lower number of steps taken to catch the evader shows better cooperation between the agents. In the simulations carried by us, this happens when the agents engulf the evader, thereby not allowing it to escape and eventually catching it.

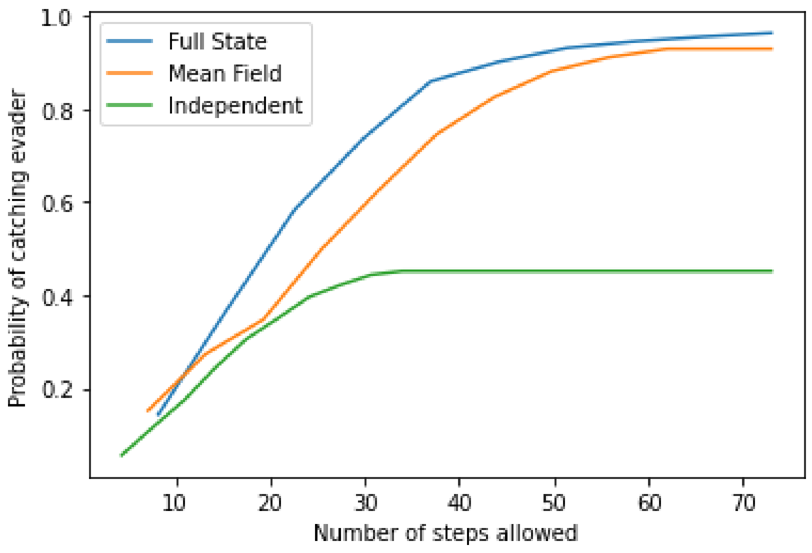


Figure 6.10: Probability of catching the evader vs max. number of time steps allowed

Figure 6.11 verifies that full state space approach and mean field approach indeed have better cooperation between the agents, in addition to better performance. In the independent agents approach, the probability remains low even for larger number of time steps. The mean field approach has a lower probability that full state space approach for lower number of steps but for higher number of steps, the probabilities are almost same.

### 6.4.5 Computation time

One of the most important aspect of the problem statement was to explore scaling the number of agents. We looked at the average Q-function calculation time by the TrainNet for all the agents. This time is expected to be different for all the three approaches since their DQN architectures are different due to different number of inputs. We varied the number of pursuer agents between 2 and 20 and had a single evader in each case. The experiments were run on **12GB NVIDIA Tesla K80 GPU** provided by Google Colab.

As expected, the average time blows up for the full state space approach as the number of pursuers are increased. This is because the input to the neural network increases proportionally, which leads to more weights and larger calculations. This is the problem highlighted by the algorithms using CLDE mechanisms. The independent approach performs the best in this regard. The mean field approach also performs reasonably good and the average time is not increasing a lot as the pursuers are increasing.

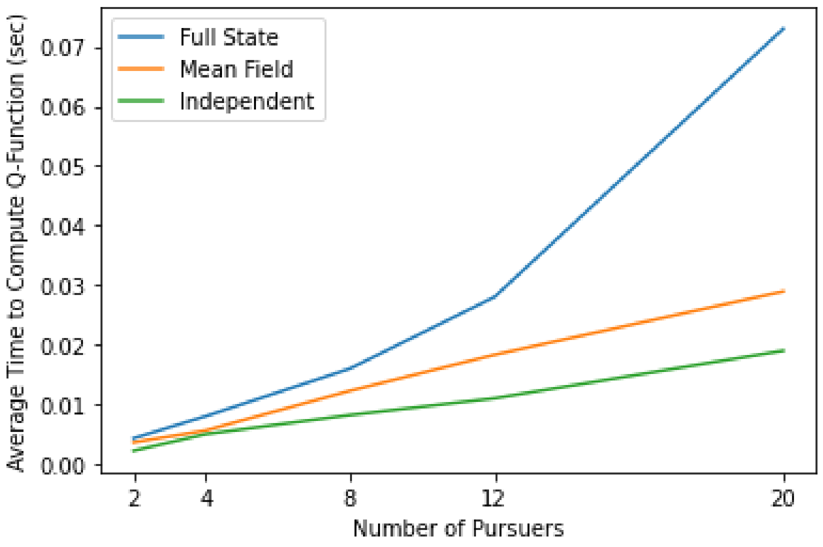


Figure 6.11: Time taken for Q-function estimation as number of pursuers are varied

### 6.4.6 Conclusion: Mean Field is the winning approach

The results discussed in this section reveal that a lot of the information about the other agents that can help in cooperation is stored in little information about the positions of the agents. This is also an experimental proof of the mean field theory in [16] that the mean field Q-function ultimately converges to the optimal Q-function. The mean field approach performs almost as good as the full state space approach, and significantly better than the independent agents approach.

## 6.5 Quantifying Cooperation

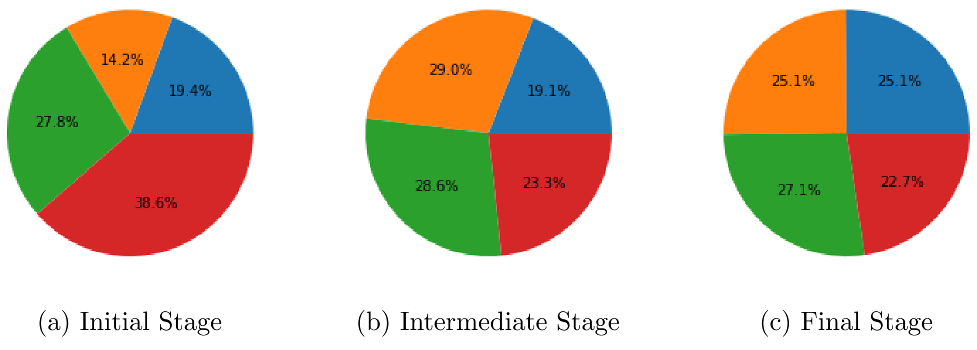
We have concluded that Mean Field approach works better compared to Full State approach due to its equivalent performance and much higher scalability. The passing of mean action of neighbours in the Neural Network in Mean Field approach leads to emergent cooperation between the agents as it performs on the same level of Full State approach and much better than Independent agents approach. This section deals with quantifying the cooperative capabilities of the agents trained using Mean Field approach.

We measure the reward value contributed by each pursuer when trying to catch the evader. Similar reward shares between the pursuers signify that they were indeed cooperating [?]. Equation 6.1 gives the cooperative ability of the pursuers or *cooperation metric* during a single episode . Here, $R_i$ is the average reward earned by $i^{th}$ pursuer over the episode given by Equation 6.2. $N$ denotes the total number of pursuers and $T_1$ and $T_2$ are the time for start and end of episodes. Lower cooperation metric means average rewards earned by all pursuers are somewhat similar signifying better cooperative capability.

$$P = \sqrt{\frac{1}{N}\sum_{i=1}^{N}(R_i - \mu)^2} \tag{6.1}$$

$$R_i = \frac{1}{T_2 - T_1}\sum_{t=T_1}^{t=T_2} r_i \tag{6.2}$$

$$\mu = \frac{1}{N}\sum_{i=1}^{N} R_i \tag{6.3}$$



(a) Initial Stage (b) Intermediate Stage (c) Final Stage

Figure 6.12: Reward share between pursuers during training

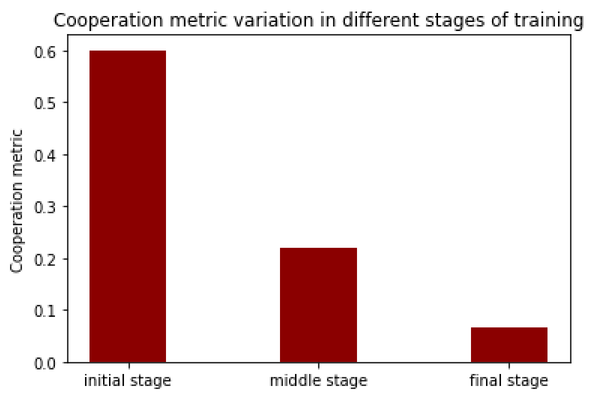


Figure 6.13: Cooperation Metric during three phases of training

For the 4 pursuers vs 1 evader experiment, we measure the reward shares for the pursuers during three phases of training. The training was done for 1000 episodes. Initial phase of training consists of first 100 episodes while final phase of training is constituted by last 100 episodes. Middle stage of training consists of episodes 450-550. Figure 6.12 shows the pursuer rewards contributed during the three stages of training. Figure 6.13 shows the cooperation metric for these three stages of training. We can see that cooperation metric decreases as training proceeds meaning that cooperative capability of pursuers improved eventually. Initial phase of training is characterized by uneven reward shares of pursuers but later on the reward shares become similar implying that agents successfully developed cooperative strategies.

# Chapter 7

# Conclusion and Future Work

## 7.1 Summary

During the course of this project, we worked on solving the multi agent problem of pursuit evasion games in a discrete environment. To differentiate our work from the previous work, we have focused on novel algorithms and cooperation between the multiple agents. We implemented the DQN algorithm with various additions such as experience replay to allow the agents to learn optimal policies in this game. The reward functions were finalised after frequent testing. Using a structured code flow and object oriented programming, we were able to implement a large number of agents ($> 10$). To allow scalability and improve cooperation between the agents, we also implemented the Mean Field reinforcement learning algorithm. Our work provides a detailed documentation of the code that we have written using state of the art Python libraries and frameworks. In order to support our choice of algorithms, we have performed a detailed analysis of the results during learning phase and testing phase and have compared the performance and feasibility of the various algorithms. In conclusion, Mean Field Reinforcement Learning combined with Deep Q-Networks outperform the standard Deep Q-Learning algorithm in terms of scalability and computational effectiveness without any deterioration in performance of agents.

## 7.2 Future Work

- We have set up the Pursuit-Evasion game on a 20*20 grid. Grid can be considered as a highly regular form of graph. The game could be set up on more generic graph based environments with different weights assigned to the edges to model congestion. These kind of environments would be hard to deal with as agents need to formulate better strategies but this would give more insight into generic form of the game.
- We observe the performance of our agents through the end result of game. Although, the policies learnt by pursuers are effective in winning the game but we

cannot comment whether the policies formulated are optimal. If some theoretical bounds can be established which can quantify the performance aspect of agents, like guarantee on win or number of moves to win, then we can comment on how close the agents are towards learning the optimal policies.

- Our approach (Mean Field + DQN) outperforms the standard DQN approaches. There are other famous algorithms like DDPG and MADDPG which perform good but face issues with scalability. Mean Field approach could be incorporated in these algorithms and compared with the Mean Field DQN approach that we worked on. This would give more clarity on the effects of Mean Field on scalability of algorithms and lead to better comparison.

# References

[1] Sutton, R. S.,  Barto, A. G. (2018). Reinforcement learning: An introduction. MIT press.

[2] R. Isaacs, "Differential Games: A Theory with Applications to Warfare and Pursuit, Control and Optimization", New York: John Wiley  Sons, 1965.

[3] Ho, Y., Bryson, A.,  Baron, S. (1965). Differential games and optimal pursuit-evasion strategies. IEEE Transactions on Automatic Control, 10(4), 385-389.

[4] Y. Ishiwaka, T. Sato, Y. Kakazu, "An approach to the pursuit problem on a heterogeneous multi-agent system using reinforcement learning", Robotics and Autonomous Systems, 43, pp. 245-256, 2003.

[5] Bilgin, A. T.,  Kadioglu-Urtis, E. (2015, July). An approach to multi-agent pursuit evasion games using reinforcement learning. In 2015 International Conference on Advanced Robotics (ICAR) (pp. 164-169). IEEE.

[6] Vidal, R., Shakernia, O., Kim, H.J., et al. 'Probabilistic pursuit-evasion games: theory, implementation, and experimental evaluation', IEEE Trans. Robotics Autom., 2002, 18, (5), pp. 662–669

[7] D. V. Dimarogonas and K. J. Kyriakopoulos. On the rendezvous problem for multiple nonholonomic agents. IEEE Transactions on Automatic Control, 52(5):916–922, 2007.

[8] Zhengyuan Zhou, Ryo Takei, Haomiao Huang, and Claire J Tomlin. A general, open-loop formulation for reach-avoid games. In IEEE 51st Annual Conference on Decision and Control (CDC), pages 6501–6506, 2012.

[9] Mnih, V., Kavukcuoglu, K., Silver, D., Graves, A., Antonoglou, I., Wierstra, D., Riedmiller, M. (2013). Playing atari with deep reinforcement learning. arXiv preprint arXiv:1312.5602.

[10] Lianmin Zheng, Jiacheng Yang, Han Cai, Weinan Zhang, Jun Wang, and Yong Yu. MA- gent: A many-agent reinforcement learning platform for artificial collective intelligence. arXiv:1712.00600, 2017.

[11] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A. Rusu, Joel Veness, Marc G. Bellemare, Alex Graves, Martin Riedmiller, Andreas K. Fidjeland, Georg Ostrovski, Stig Petersen, Charles Beattie, Amir Sadik, Ioannis Antonoglou, Helen King,

Dharshan Ku- maran, Daan Wierstra, Shane Legg, and Demis Hassabis. Human-level control through deep reinforcement learning. Nature, 518:529–533, 2015.

[12] J. Zhu, W. Zou and Z. Zhu, ”Learning Evasion Strategy in Pursuit-Evasion by Deep Q-network,” 2018 24th International Conference on Pattern Recognition (ICPR), Beijing, 2018, pp. 67-72, doi: 10.1109/ICPR.2018.8546182.

[13] Timothy P Lillicrap, Jonathan J Hunt, Alexander Pritzel, Nicolas Heess, Tom Erez, Yuval Tassa, David Silver, and Daan Wierstra. Continuous control with deep reinforcement learning. arXiv:1509.02971, 2015.

[14] Lowe, R., Wu, Y., Tamar, A., et al.: ‘Multi-agent actor-critic for mixed cooperative-competitive environments’. Advances in Neural Information Processing Systems, Long Beach, CA, USA, 2017, pp. 6379–6390

[15] Sukhbaatar, Sainbayar, and Rob Fergus. ”Learning multiagent communication with backpropagation.” Advances in neural information processing systems. 2016.

[16] Yaodong Yang, Rui Luo, Minne Li, Ming Zhou, Weinan Zhang, and Jun Wang. Mean field multi-agent reinforcement learning. arXiv:1802.05438, 2018.

[17] Subramanian, S. G., Poupart, P., Taylor, M. E., Hegde, N. (2020). Multi Type Mean Field Reinforcement Learning. arXiv preprint arXiv:2002.02513.

[18] Zheng, L., Yang, J., Cai, H., Zhou, M., Zhang, W., Wang, J., Yu, Y. (2018). MAgent: A Many-Agent Reinforcement Learning Platform for Artificial Collective Intelligence. Proceedings of the AAAI Conference on Artificial Intelligence, 32(1). Retrieved from https://ojs.aaai.org/index.php/AAAI/article/view/11371