Evaluation of Multi-Agent Deep Reinforcement Learning Algorithms in the Pursuit-Evasion Environment

DHAVAL SALWALA

MSc. (Hons)

College of Engineering and Informatics

National University of Ireland, Galway

A thesis submitted in fulfilment of the
requirements for the degree of

Master of Science - Data Analytics

August 2019

Thesis Supervisor: Dr. Frank Glavin

Abstract

The modern world is desperately trying to replace every aspect of the human world with Artificial Intelligence (AI). The powerful AI systems are already taking over most parts of everyday life. Machine Learning can be used to solve a variety of problems - navigation, engineering projects, pre-disaster warning, healthcare and there is no end. Research has shown that doing a task in co-operation with other agents greatly reduce the complexity of the problem and improve both quality & speed of learning. Cooperative learning as observed in the human and animal world shows acquisition of sound knowledge and learning amongst agents which may result in higher effectiveness compared to individual learning. Hence, this thesis tries to evaluate the performance of multi-agents in a co-operative environment setup.

Reinforcement Learning (RL) is the form of machine learning that discovers the uncharted territory and learns from its mistakes. RL is attributed to the study of goal-directed behaviour. Fundamentally, an RL agent decides what actions need to be selected to achieve the goal using the time-delayed rewards. However, it has also been proven to be an extremely complicated and time-consuming task to convert the goal of the problem into a reward signal. A notable part of the RL literature has focused on single agents, nonetheless, most realistic situations involve the presence of multiple agents. Knowledge sharing is one of the most important traits humans exploit to tackle any complex problem. It is possible to achieve similar performance in machines via Multi-Agent Reinforcement Learning (MARL). MARL poses several challenges in terms of heterogeneity of agents and sharing of reward signals. This thesis tries to learn the behaviour of a multi-agent system in a cooperative way where agents can share experience and knowledge among them. In contradiction to typical single-agent approaches where the learning is monotonous, the architecture presented in this thesis uses a de-centralised parameter sharing across multiple agents.

This thesis demonstrates the learning course of multi-agent deep reinforcement learning algorithms in a setup called Pursuit Evasion. Three multi-agent versions of the RL algorithms have been explored in this research viz. Deep Q networks (DQN), Reinforce and Advantage Actor-Critic (A2C). The evaluation results reveal that Reinforce has a better convergence rate in capturing the spatial correlation of the domain using deep learning. A2C is a slow learner and could do better with parameter refinements. In the experiments performed in this thesis, DQN did not perform well owing to the non-stationarity of the domain.

Declaration

I declare that this thesis was composed by myself, that the work contained herein is my own except where explicitly stated otherwise in the text, and that this work has not been submitted for any other degree or professional qualification except as specified.

_____

Dhaval Salwala
August 2019

Contents

List of Figures

List of Algorithms

1. Introduction

1.1 Motivation

In recent years, Reinforcement Learning (RL) has achieved a remarkable performance in different machine learning tasks. It has pushed the boundaries of AI beyond the possibility one could have thought in the past [Silver et al., 2017a]. The main aim of RL is to let the agent learn the changing circumstances of the environment using raw inputs and time-delayed rewards. We saw a terrific performance of RL in different Atari 2600 [Minh 2013, 2015] games and DeepMind's AlphaGo Move 37 that beat the world's best Go player [Silver et al., 2016]. AlphaGo learned its moves by analysing millions of episodes played by humans and getting help from hand-coded heuristics [Silver et al., 2016]. Then came AlphaGo Zero, which out-played AlphaGo without ever having seen a human play the game [Silver et al., 2017b]. AlphaGo Zero just took 40 days of self-training to outperform its older version and defeated the world number one Ke Jie. Still, researchers are looking for innovations in setting out to build a new version of the A.I. that learn on its own in a most efficient way. Infrastructure has always been the roadblock to train such a System. AlphaGo Zero was trained on 5000 TPUs [Silver et al., 2017b]. This is a huge amount of computing power. In order to overcome this problem, researchers are finding an alternative way of training where Multi-Agent Setup can play an important role in bringing revolution to the AI innovations.

With the advent of toolkits like Unity ML-Agent [Juliani et al., 2018] and OpenAI Gym [Brockman et al., 2016], we can foresee many powerful capabilities of RL in the real world. These toolkits allow us to test a prototype in a simulated environment. Simulated environments are very useful for testing of any RL scenarios and they provide safe passage of training [Sutton and Barto, 1988] before it can be put forward into practice. After successful completion of training and all black box testing, the idea can be applied in real-life scenarios or at least a part of it. For example, researchers at the Computer Vision Center have developed a simulation platform specifically designed for testing autonomous cars – CARLA [Dosovitskiy et al., 2017], while Tesla, an independent company developing its own self-driving car technology – Tesla Autopilot [Tesla, 2015], has managed significant achievement in self-driving technology.

Nonetheless, a significant amount of work has been carried out in the area of cooperative learning. Researchers have been able to significantly reduce efforts in the manual annotation of social signals in multi-modal corpora of considerable size using Cooperative Machine Learning (CML) [Wagner et al., 2018]. In another instance, [Vidhate and Kulkarni, 2012] proposed a design for building a cooperative machine learning system that contains two or more machine learners that cooperate.

1.2 Purpose

The thesis will present the performance of three different multi-agent reinforcement learning algorithm in a Pursuit-Evasion environment. The key innovation is finding a deep learning architecture for agents and knowledge sharing strategy amongst them to automatically form a learning curriculum. The findings of the research will be compared among algorithms to evaluate their ability to grasp the environment. Both, training and evaluation of the learned policies, will be conducted in a simulated learning environment.

The goal of this thesis is to study whether reinforcement learning agents can learn to operate co-operatively in an environment using only sparse reward signals.

1.3 Scope

The simulated environment will make use of the Multi-Agent Deep Reinforcement Learning (MADRL) environment created by Stanford Intelligent Systems Laboratory [Gupta et al., 2017]. This setup internally uses OpenAI Gym [Brockman et al., 2016] toolkit to emulate various Multi-Agent scenarios. The aim is to train our agents in a simulated Evasion-Pursuit terrain, which would be otherwise be tedious and resource intensive in the real world. The environment and action spaces are discrete.

The training will be conducted using a single machine with a desktop variant of the 8th Generation 6-core Intel Core i5-8400 processor having 8GB Ram. The training experiment will be conducted using CPU only. Simulation is carried out with 20 adversaries and 10 allies in a 16x16 2D domain.

1.4 Outline

The following section serves to introduce the reader with the understanding of Markov Decision Process (MDP) and Reinforcement Learning (RL). RL approaches like q-learning and policy gradient have been discussed. The third section introduces the methodology of building an environment, bootstrapping agents, understanding model architecture as well as de-centralized parameter setup. In addition to this, section 4 introduces the three multi-agent version of the RL algorithms used in this thesis. The evaluation results are discussed in section 5. Finally, we summarize and conclude the research in section 6.

2. Background

1.2 Markov Decision Processes

Markov Decision Process (MDP) is a mathematical framework behind Reinforcement Learning (RL). MDPs are useful in optimization tasks, control and robotics. It is based on Markov property and Markov chain. The stochastic process has a Markov property if its next state depends only upon its current state i.e. conditional probability distribution of the next stage of the process depends only upon the present state [Silver, 2015].

Types of RL problems that can be formalised as MDP:

- Optimal control primarily deals with continuous MDPs
- Partially observable problems can be converted into MDPs
- Bandits are MDPs with one state

Problem Formulation

A Markov chain is a stochastic process with the Markov property. A Markov chain can traverse between states and each transition depends on its transitional probability P [Silver, 2015].

Transition probabilities P from all states s to all successor states s' can be defined in the matrix form as follows:

$$P = from\begin{pmatrix} P_{11} & \cdots & P_{1n} \\ \vdots & \ddots & \vdots \\ P_{n1} & \cdots & P_{nn} \end{pmatrix}$$

*Each row of the matrix sums to 1

A Markov reward process (MRP) is a Markov chain with values.

MRP [S, P, R, γ] is defined as [Silver, 2015],

1) S is a finite set of states
2) P is a state transition probability matrix $P_a(s, s') = \Pr(s_{t+1} = s' | s_t = s)$

3) R is a reward function. $R_s = \mathrm{E}[R_{t+1}|S_t = s]$

4) γ is a discount factor $\gamma \in [0,1]$

The agent's goal is to maximize the discounted sum of its future rewards, R_t, defined by,

$$R_t = R_{t+1} + \gamma R_{t+2} + \cdots = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1}$$

Equation 1: discounted sum of its future rewards

γ is the discount factor that is added to give more importance to immediate rewards with respect to future ones. Cases where it may not be beneficial to use discount factor, γ is 1. Whereas in the financial system, immediate rewards may be more interesting than delayed ones. It also avoids infinite returns in cyclic Markov processes.

Markov Decision Process (MDP) is an extension of Markov chains with two additions: choosing actions and rewards. It helps to determine the ideal behaviour of an agent within the environment in order to maximize the model's ability to achieve a certain state. The goal is to determine the optimised policy, which the agent aim to take on the environment for the long-term maximum reward. The entire sequence depends on the feedback from the environment for every agent's interaction [Sutton and Barto, 1988].

MDP Formal Definition [S, A, P, R, γ],

1) S is set of all possible states: $S_i \in S | i = 1 \dots n$

2) A is set of all actions $a_i \in A | i = 1 \dots m$

3) P is the probability that action a in state s at time t will lead to state s' at time $t + 1$.

$$P_a(s, s') = \Pr(s_{t+1} = s | s^t = s, a^t = a)$$

4) $R_a(s, s')$ is the immediate reward received after transiting from state s to state s', due to action a.
5) γ is a discount factor, $\gamma \epsilon [0,1]$

Understanding the model

From the above formal definition, the information about state s is available to the agent. An Agent is awarded a reward R which is dependent on state s and action a. $s_{t+1} = s'$ is the future state that we intend to get. $[s1 \ldots . st]$ are all the intermediate states in the state's history. P is the transitional probability of getting into state s' from state s via action a. We start in some state s, we act a, we reach state s' by the probability of P. Thus, the new state depends only on the current state and the action responsible for the transition. It has no connection with states happened before. MDP is a memoryless model.

MDP assumes full accountability of the environment at any given state. The Markov property anytime can be defined as,

$$P(S_{t+1} = s' \mid S_t = s, A^t = a) = P(S_{t+1} = s' \mid S_1 = s_1, A_1 = a_1 \ldots S_t = s_t, A_t = a_t)$$

Equation 2: Probability according to Markov Property

MDPs evaluate and use policies to decide what action to take at each state. A policy π represents a solution to the MDP problem. It defines what action to take in each state. Optimal policy yields the highest utility.

$$\pi(S) \rightarrow a$$

$\pi^*(S)$: Optimal Policy

Following the policy π, the state value function $v_\pi(s)$ is defined as the expected sum of discounted reward when starting from state s. It gives the long-term value of state s.

$$v_\pi(s) = \mathrm{E}_\pi[G_t | S_t = s]$$

Equation 3: Value function as the expected return

Similarly, Following the policy π, an action-value function $q_\pi(s, a)$ is defined as the expected return when starting from state S and taking the action a:

$$q_\pi(s, a) = \mathrm{E}_\pi[G_t | S_t = s, A_t = a]$$

Equation 4: Action Value function as the expected return

Above state value function and action value function can be decomposed into two parts which leads to the Bellman equation [Bellman, 1954] [Bellman, 1956] for solving MDP problem.

- Immediate Reward: R_{t+1}
- Discounted value of next state: $\gamma v(S_{t+1})$

$$v_\pi(s) = \mathrm{E}_\pi[R_{t+1} + \gamma v_\pi(S_{t+1}) | S_t = s]$$

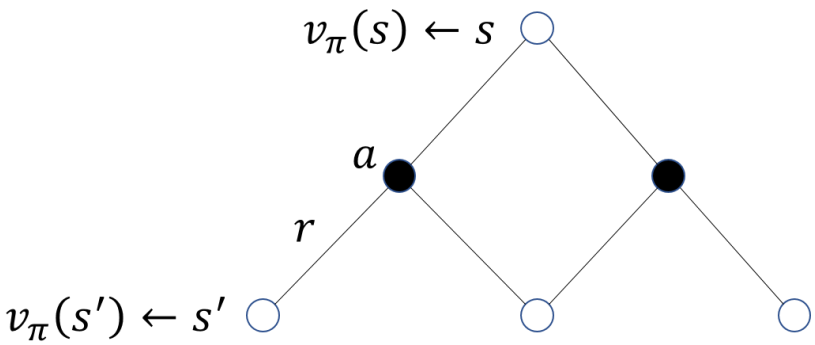
$$q_\pi(s, a) = \mathrm{E}_\pi[R_{t+1} + \gamma q_\pi(S_{t+1}, A_{t+1}) | S_t = s, A_t = a]$$

Equation 5: Bellman Expectation Equation

2.1.1 Solving Markov Decision Processes

First let us expand and understand the Bellman equation [Bellman, 1954] [Bellman, 1956] as applied in Equation 5. It is known that the total discounted reward is the expected immediate reward and all future discounted rewards when following the same policy π [Silver, 2015].

Bellman Expectation Equation for $v_\pi(s)$:



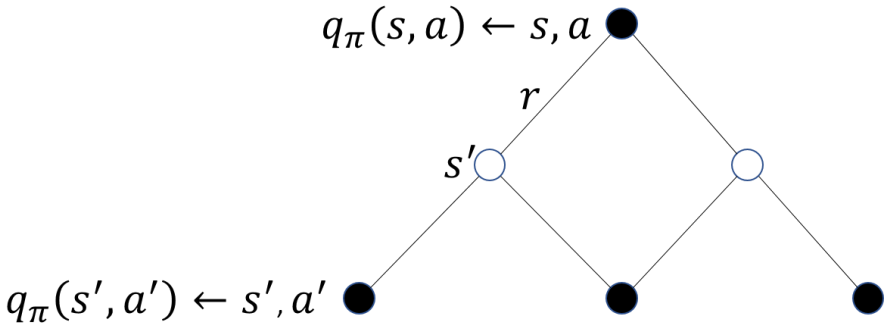
Explanations:

1) We have a value function at a state S.
2) Then we consider a policy π that takes two stages look ahead and considers all the actions we might take (dark circles).
3) We take into notice all the responses of the environment against the corresponding action (empty circles).
4) For each of the environment responses (empty circles), we try to find out the goodness i.e. maximum reward possible for being in the stage S'.
5) We do an average over policy and all the transition probability. All of them together gives us the value $v_\pi(s)$ of being in the stage S.

$$v_\pi(s) = \sum_{a \in A} \pi(a, s)(R_s^a + \gamma \sum_{s' \in S} P_{ss'}^a v_\pi(s'))$$

Equation 6: Bellman Expectation Equation for State Value function

Bellman Expectation Equation for $q_\pi(s, a)$:



Explanations:

1) We take an action A in stage s as per policy π.
2) For each of the environment responses (empty circles), we try to find out the goodness i.e. maximum reward possible for being in the stage S'.
3) Then we consider considers all the actions we might take (dark circles). We take into notice all the responses of the environment against the corresponding action (empty circles).

4) We do an average over policy and all the transition probability. All of them together gives us the $q_\pi(s,a)$ of being in the stage s.

$$q_\pi(s,a) = R_s^a + \gamma \sum_{s' \in S} P_{ss'}^a \sum_{a' \in A} \pi(a'|s') q_\pi(s',a'))$$

Equation 7: Bellman Expectation Equation for Action Value function

Goal of any RL problem is to find the policy which maximise the expected future reward. For any Markov Decision Process, there exist an optimal policy which maximises state / action value function over all policies.

$$v_{\pi*}(s) = v_*(s), q_{\pi*}(s,a) = q_*(s,a)$$

An optimal policy can be found by maximising over $q_*(s,a)$.

$$\pi^*(a|s) = \begin{cases} 1, & if\ a = \underset{a \epsilon A}{\operatorname{argmax}}\, q_*(s,a) \\ 0, & otherwise \end{cases}$$

Hence, re-writing Bellman Expectation Equation for an optimal policy

$$v_*(s) = \max_{a \epsilon A} R_s^a + \gamma \sum_{s' \in S} P_{ss'}^a v_*(s')$$

$$q_*(s,a) = R_s^a + \gamma \sum_{s' \in S} P_{ss'}^a \max_{a'} q_*(s',a')$$

Equation 8: Bellman Optimality Equation

Many iterative techniques exist for solving a Bellman Optimality Equation like Value Iteration, Policy Iteration, Q-learning and SARSA [Sutton and Barto, 1988] .

2.2 Reinforcement Learning

Reinforcement learning (RL) is all about making decisions based on reward and mistakes. The agent interacts exclusively with the subject at every step and acts based on a defined policy to maximise the future outcome [Sutton and Barto, 2018]. The agent is not informed about which actions to take but instead must try and interact with every element of the environment to yield maximum future rewards. The agent is a certain entity which has many actions to choose from. The actions can be discrete or continuous. States contain information about the current environment. Below is the schematic diagram explaining state-actor-environment relation in RL.

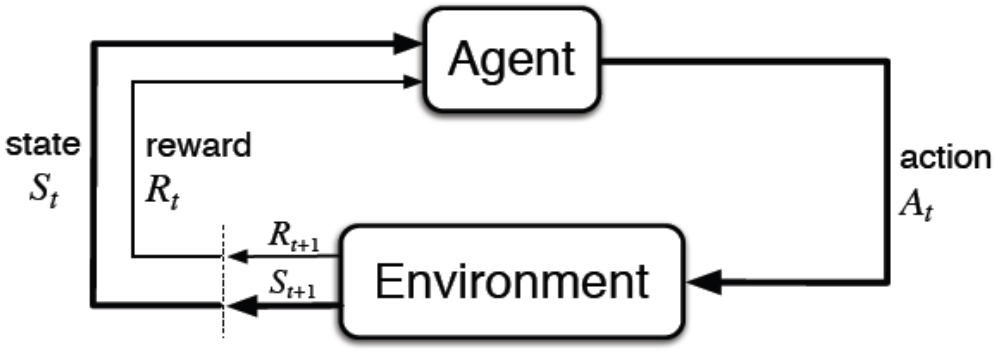


Figure 1: The agent–environment interaction in a Markov decision process [Sutton and Barto, 2018]

Sutton and Barto, 2018] states that MDPs are a mathematically idealized form of the reinforcement learning problem that forms the framework of the precise theoretical statements. MDPs are meant to be a straightforward framing of the problem of learning from interaction to achieve a goal. The problem formulation boils down to the three basic signals that are passed from the environment to the agent and vice versa as shown in Figure 1.

- Actions
- States
- Rewards

Initially, the agent has no information about the environment and takes random actions to learn what states result in what rewards. This is an exploration stage and agent must comprehend to maximise its learning without losing too much reward along the way. Later, the agent exploits the same learning to gain more long-term rewards. The reward received at every step can be both negative and positive. The agent's goal is to earn maximum possible rewards even when rewards are not available early. Hence, it must learn to avoid low and negative reward signals. The use of a reward signal to formalize the idea of a goal is one of the most distinctive features of RL. As per [Sutton and Barto, 2018], RL is based on the idea of the reward hypothesis. A goal is the maximisation of

cumulative rewards. The agent evaluates the current rewards, updates its knowledge and takes next action based on the new state. This process continues until the environment throws a terminal state. This is the end of a single episode.

RL can be used to solve a range of problems such as navigation, automation, learning features of an uncharted environment as well as achieving maximum performance in a complex video game [Silver, 2015].

Major Components of an RL Agent [Sutton and Barto, 2018]:

- Policy: It represents an Agent's behaviour. It is a map from state to actions. A policy can be of two types.
  - Deterministic $a = \pi(s)$
  - Stochastic $\pi(a|s) = P[A_t = a|S_t = s]$

- Value Function: It predicts future rewards. Evaluates how good/bad is the current state.

$$v_\pi(s) = \mathrm{E}_\pi[R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + I|S_t = s]$$

Equation 9: Value function as the sum of expected returns

- Model: It predicts what the environment will do next.

$$P^a_{ss'} = \mathrm{P}(S_{t+1} = s'|S_t = s, A_t = a)$$
$$R_a(s) = \mathrm{E}(R_{t+1}|S_t = s, A_t = a)$$

Equation 10: Total Expected Returns

Based on the above parameters, RL agents fall into different categories as shown in Figure 2.

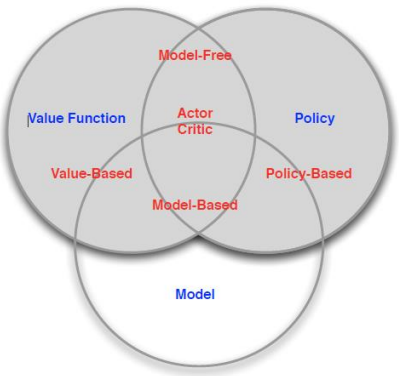


Figure 2: Reinforcement Learning agent taxonomy [Silver, 2015]

2.2.1 Policy Iteration

Iterative RL techniques like value iteration and policy iteration can be solved using Dynamic Programming (DP). DP is more applicable to the problem where the environment is finite. In a finite state MDP, state, action, and reward sets, S, A, and R are finite, and that its dynamics are given by a set of probabilities $P(s', r \mid s, a)$. The key idea of RL, in general is the use of value functions to organize and structure the search for good policies.

In MDP, the Bellman equation gives recursive decomposition and Value function stores and reuse solutions. These are the two properties that need to be satisfied to use Dynamic Programming [Silver, 2015].

Policy Iteration can be solved using a Bellman expectation Equation 5

Given a policy π,

- Evaluate the policy π

$$v_\pi(s) = \mathrm{E}[R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + I | S_t = s]$$

- Improve the policy by acting greedily with respect to v_π

$$\pi' = greedy(v_\pi)$$

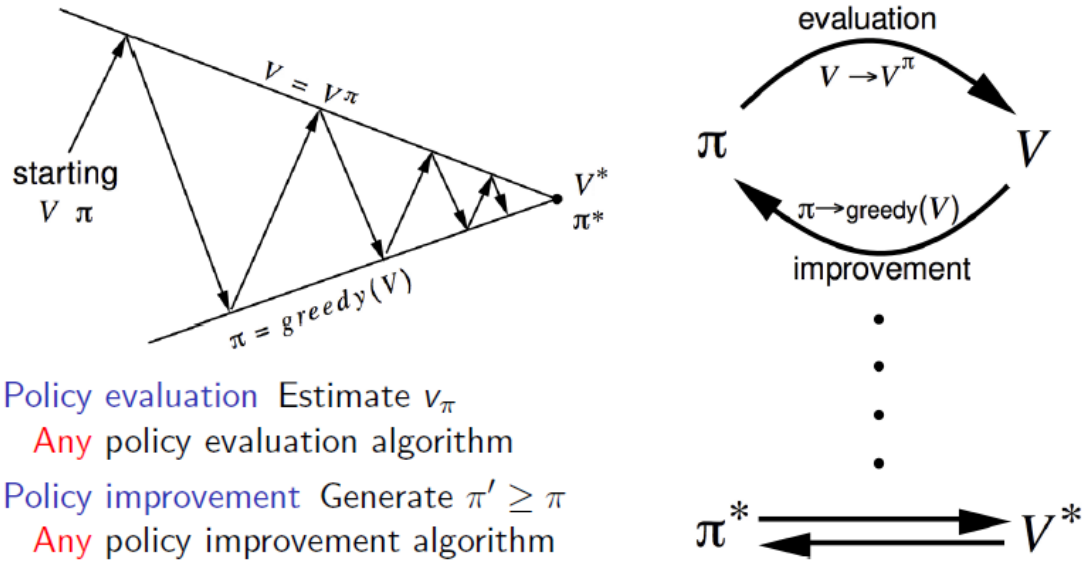


Figure 3: Generalised Policy Iteration[Silver, 2015]

2.2.2 Value Iteration

One drawback of policy iteration is that is we must be able to find out when we need to stop evaluating policy. We can wait for the exact conversion or stop early without affecting the convergence state of policy iteration. One way of achieving these is to stop after just one sweep. This is called value iteration. Value iteration can be solved using a Bellman optimality Equation 8.

Principle of Optimality [Silver, 2015]:

A policy $\pi(a|s)$ achieves the optimal value from state s, $v_\pi(s) = v_*(s)$, if and only if,

- For any state s' reachable from s
- π achieves the optimal value from state s', $v_\pi(s') = v_*(s')$

If we know the solution to the sub-problem $v_*(s')$, then the solution $v_*(s)$ can be found using the Equation 8.

2.3 Model Free Learning

In the last section, we saw dynamic programming can be used to solve the finite state MDP problem. We do not assume complete knowledge of the environment i.e. we have no idea about the transitional probabilities and rewards. Model-free RL algorithms learn directly from the episodes of experience. There are variants of model-free learning like one step learning or n-episodic learning or n-step learning. Some techniques try to learn experience without bootstrapping while some do not.

The two most common methods for model free learning are,

- Monte Carlo Learning
- Temporal Difference Learning

2.3.1 Monte Carlo Learning

Monte Carlo (MC) belongs to the family of model-free learning RL algorithms that learns from entire episodes of experience. The return after taking an action in one state now depends on the actions taken in later states in the same episode. Hence, there is no bootstrapping. A model is required only to generate sample transitions and not the complete transitional probability distributions. Such experience sequences consist of states, actions, and rewards from actual or simulated interaction with an environment. It uses an empirical mean of the episodic returns to estimate the value [Sutton and Barto, 2018]. All episodes must terminate in MC. Two variants to evaluate a policy are First-Visit

Monte-Carlo Policy Evaluation or Every-Visit Monte-Carlo Policy Evaluation [Silver, 2015].

Evaluation Strategy [Silver, 2015], [Sutton and Barto, 2018]:

1. Learn v_π from the series of episodic experiences under policy π.

$$S_1, A_1, R_2, I, S_k \sim \pi$$

2. Calculate total returns

$$G_t = R_{t+1} + \gamma R_{t+2} \; + \; I \; + \; \gamma^{T-1} R_T$$

3. MC policy evaluation uses empirical mean return as per Equation 3. We can only compute this value after we have completed all the episodes. MC uses Incremental Mean equation to update the value incrementally. It may be possible to forget old episodes in case of non-stationary mean.

$$V(S_t) \; \leftarrow \; V(S_t) + \frac{1}{N(S_t)}(G_t - V(S_t))$$

$$\text{Where, } N(S_t) \; \leftarrow N(S_t) + 1$$

Equation 11: Incremental MC Updates

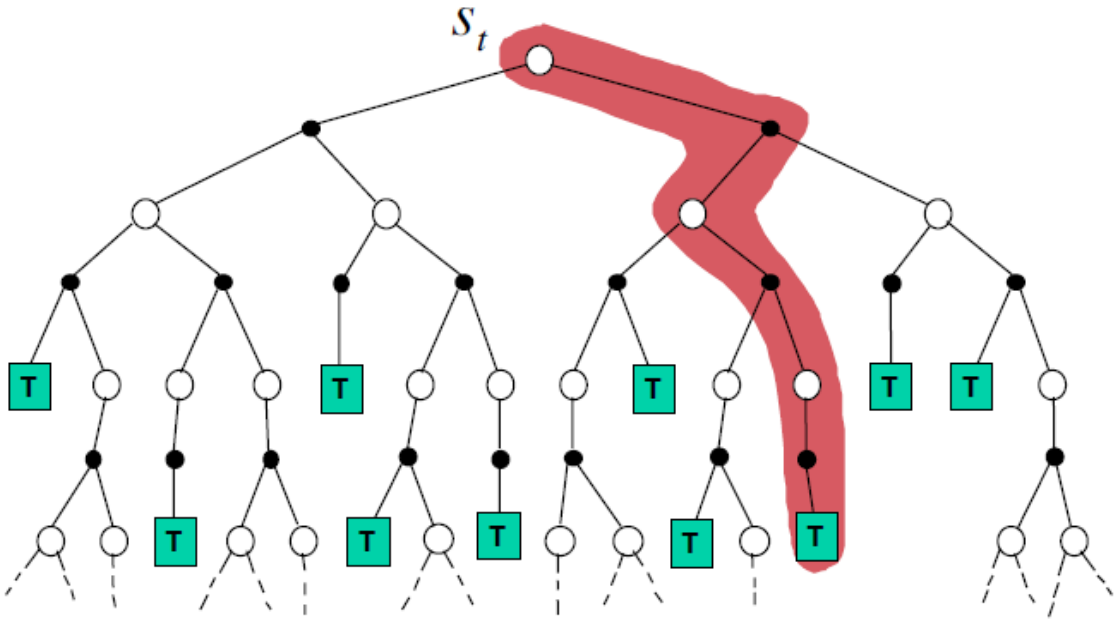


Figure 4: MC Backup [Silver, 2015]

4. For First-Visit Monte-Carlo, counter $N(S_t)$ is incremented for only fist visit of state S_t. While it is raised by 1 for every visit in Every-Visit Monte-Carlo Policy Evaluation.

5. MC converges to solution with minimum mean-squared error.

$$\sum_{k=1}^{K} \sum_{t=1}^{T_k} (G_t^k - V(s_t^k))^2$$

6. This thesis later discusses on Monte Carlo Reinforce Policy Gradient learning [Silver, 2015].

2.3.2 Temporal Difference Learning

Temporal Difference (TD) is a model-free. It learns from incomplete episodes. TD updates the value function at each n step. Here number n is the n-step return into the future. Thus, it learns by bootstrapping. While TD can learn online after every step without the outcome, MC must wait until the end of the episode before calculating an average return. Thus, TD works very well in non-stationary environments.

Evaluation Strategy [Silver, 2015]

1. The simplest temporal-difference learning algorithm TD(0) where n=0, update the value $V(S_t)$ towards an estimated return $R_{t+1} + \gamma V(S_{t+1})$.

$$V(S_t) \leftarrow V(S_t) + \alpha(R_{t+1} + \gamma V(S_{t+1}) - V(S_t))$$

Equation 12: TD Update

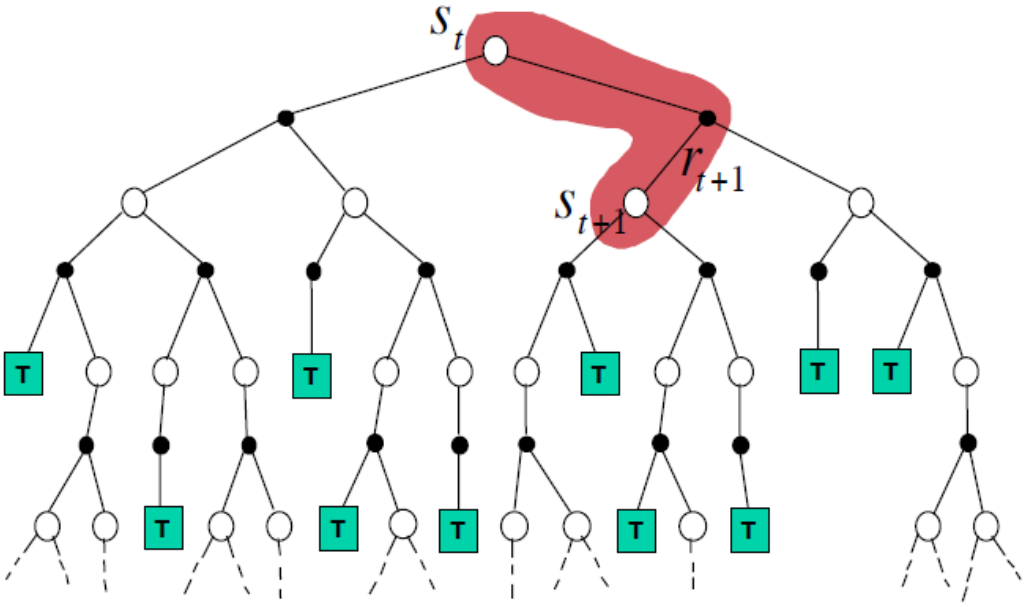


Figure 5: Temporal Difference Backup [Silver, 2015]

2. In above Equation 12,
   a. $R_{t+1} + \gamma V(S_{t+1})$ is called the TD Target while
   b. $R_{t+1} + \gamma V(S_{t+1}) - V(S_t)$ is called the TD error.

3. TD(0) converges to solution of max likelihood Markov model.

$$R_s^a = \frac{1}{N(s,a)} \sum_{k=1}^{K} \sum_{t=1}^{T_k} 1(s_t^k, a_t^k = s, a) r_t^k$$

4. N-step return into the future,

$$V(S_t) \leftarrow V(S_t) + \alpha(G_t^{(n)} - V(S_t))$$

Equation 13: n-step TD update

5. This thesis later discusses the deep version of the TD(0) Q learning [Sutton and Barto, 1988] also called DQN [Mnih et al., 2015].

2.4 Function Approximation

One of the goals of the field of RL is to solve complex tasks from unprocessed, high-dimensional input. RL has recently been applied to solve a variety of challenges ranging from playing the game of GO [Silver et al., 2016] to the robotics [Levine et al., 2015]. RL is emerging as a prominent approach to cooling large scale data systems [Evans and Gao, 2016]. But there's always been a hurdle to scale up model-free methods to handle large neural network policies and value functions. Unfortunately, traditional reinforcement learning approaches such as Q-Learning tends to perform poorly when there is a high-dimensional observation space. Earlier RL algorithms are known to overestimate action values under certain conditions [van Hasselt et al., 2015]. There are just too many states and actions which dampen the speed of learning [Silver, 2015]. Q-Learning is helpful only in the case of the finite-state model because the size of Q Table will increase exponentially with the increase in action spaces [Finnman and Winberg, 2016]. The deep neural network provides flexible function approximation with the potential for a low asymptotic approximation error and prevents the harmful effects of noise [van Hasselt et al., 2015].

- Update parameter w using MC or TD learning
- There are many function approximator but in this thesis, we will be using Neural Network to estimate q values and policies.
- $\hat{v}(s,w) \approx v_\pi(s)$
- $\hat{q}(s,a,w) \approx q_\pi(s,a)$

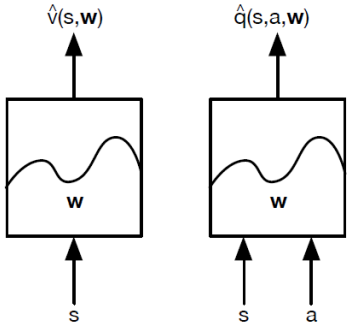


Figure 6: Value Function Approximation [Silver, 2015]

2.4.1 Deep Q Network

Q-learning is a temporal difference method to incrementally estimate the value of taking action from a given state by continuously updating Q-values. The deep Q-network [Mnih et al., 2015] uses a neural network to approximate the state-action value function.

$$Q(s,a) = \max_{\pi} \mathrm{E}[r_t + \gamma r_{t+1} + I \,|s_t = s, a_t = a, \pi]$$

DQN makes use of experience replay database which stores the agent's experience $e_t = (s_t,\ a_t,\ r_t,\ s_{t+1})$ to reduce correlations between experiences. The experience is unique tuple that consists of the current state s_t, current action a_t, next state s_{t+1} and a received reward r_t. The update to neural network weights is carried out by minimising a loss function based on a TD update Δw [Simonini, 2018].

Learning rate

$$\Delta w = \alpha[(R + \gamma max_a \hat{Q}(s', a, w)) - \hat{Q}(s', a, w)]\nabla_w \hat{Q}(s', a, w)$$

Weight update — TD Error — Current Gradient

Figure 7: Weights Update in DQN

DQN cannot easily handle continuous action spaces. There are many improvements to DQN [Mnih et al., 2015]. In this thesis, we are using an enhanced version of DQN with target network and double DQNs. In theory, we calculate the loss by taking the difference between TD target and current Q value. The TD Target is calculated using the Bellman Equation 7. It is the reward of taking that action at that state plus the discounted highest Q value for the next state. We use the neural network to estimate the next Q for the next state. Since we same weights to estimates both target and Q values, there is a big correlation between the TD target and the parameters (w) we are changing. This leads to a big inconsistency in the learning process.

Learning rate

$$\Delta w = \alpha[(R + \gamma max_a \hat{Q}(s', a, w')) - \hat{Q}(s', a, w)]\nabla_w \hat{Q}(s', a, w)$$

Weight update — TD Error — Current Gradient

Weights of Target network
w' <- w *every fixed interval*

Figure 8: Fixed-Q Target [Simonini, 2018]

Instead, we can use a separate (target) network with a fixed parameter for estimating the TD target. At every fixed interval, we copy the weights from the DQN network to the target network [van Hasselt et al., 2015].

Double DQNs was introduced in 2015 by Hado van Hasselt [van Hasselt et al., 2015] to handle the overestimation of Q-values by the neural network. Here, we use Q network to select what is the best action to take for the next state and use the target network to calculate the target Q value of taking that action at the next state. This helps us to train faster and have more stable learning.

$$\underbrace{Q(s,a)}_{\text{TD Target}} = r(s,a) + \gamma Q(\underbrace{s', \underbrace{argmax_a Q(s',a)}_{\text{Choose next action using DQN}}}_{\text{Calculate Q value of taking that action using the Target Network}})$$

Figure 9: Double DQN [Simonini, 2018]

2.5 Policy Gradient Learning

Policy-based reinforcement learning algorithms directly map the state to action via a policy function. Most basic policy-based techniques don't care about value function and learn directly by optimising a policy function π. While advanced policy-based method viz. A2C uses value function as a critic to guide the policy head. We directly parametrise the policy function π. This π outputs a probability distribution of actions. Policy improvement is an optimization problem and is used to update the parameter θ.

$$\pi_\theta(s,a) = P[a \mid s, \theta]$$

Equation 14: Probability of taking action 'a' given state 's' with parameters theta.[Silver, 2015]

Policy-based methods have better convergence properties and they are effective in high-dimensional or continuous action spaces. They can learn stochastic policy as well [Silver, 2015].

Policy-based methods try to maximise a score function $J(\theta)$ by ascending the gradient of the policy, w.r.t. parameters θ.

$$\Delta\theta = \alpha \nabla_\theta J(\theta)$$

Where,

$\nabla_\theta J(\theta) = \begin{pmatrix} \frac{\partial J(\theta)}{\partial \theta_1} \dots \\ \dots \frac{\partial J(\theta)}{\partial \theta_n} \end{pmatrix}$ *is a policy gradient*

$J(\theta) = E_\pi[R(\tau)]$ *is a score function*

$R(\tau)$ *is expeceed future reward*

E_π *is expected given policy*

α *is a step size parameter*

Equation 15: Policy Gradient Update [Silver, 2015]

Using likelihood ratio trick [Ecoffet, 2018], we can write the policy gradient as,

$$\nabla_\theta J(\theta) = E_\pi[\nabla_\theta (log\pi(s, a, \theta)) R(\tau)]$$

Equation 16: Policy Gradients [Simonini, 2018]

$R(\tau)$ is a total expected reward. We push the gradient in the direction of the actions that have the highest return i.e. push up the probability of taking these actions if $R(\tau)$ is higher and vice versa [Simonini, 2018].

2.5.1 Reinforce – Monte Carlo Policy Gradient

Reinforce measures the expected reward R_t for a full trajectory and use that to update the policy gradient by stochastic gradient ascent. Hence, it is called the Monte Carlo method.

A variation of Reinforce exists that subtract a baseline value from the return R_t to reduce the variance without changing expectation. A good baseline is the state value function $V^{\pi_\theta}(s)$. Rewriting policy gradient Equation 16 using the advantage function,

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

$$\nabla_\theta J(\theta) = E_\pi[\nabla_\theta (log\pi(s, a, \theta)) A^{\pi_\theta}(s, a)]$$

Equation 17: Reinforce with Baseline

2.5.2 Advantage Actor-Critic Policy Gradient

Advantage Actor-Critic is a synchronous implementation of the asynchronous advantage actor-critic (A3C) proposed in [Mnih et al., 2016]. It is an on-policy RL method that makes use of separate components – policy model and value function. As stated in [Mnih et al., 2016], knowing value function can be useful in reducing the gradient variance

when compared to the vanilla approaches like Reinforce. Value function can tell us how good a policy is. Actor-Critic takes advantage of this fact to define a parameterised value function in addition to the policy. A2C follows an approximate policy gradient. The critic solves a policy evaluation problem via either MC or TD update.

It maintains two sets of parameters [Silver, 2015],

a) Updates action-value function parameters w for the critic network.
b) Updates policy parameters θ for the actor-network, in the direction, suggested by the critic.

In A2C, we don't wait until the end of the episode as we do in REINFORCE. At every step, A2C calculates updates as in TD(0) update. The agent bootstraps its way through the time steps. The degree of bootstrapping is configurable.

Two learning rates, α and β, are predefined for policy and value function.

$$\Delta\theta = \alpha * \nabla_\theta * (log\pi_\theta(s, a, \theta)) * Q_w(s, a)$$

Equation 18: A2C Policy Update

$$\Delta w = \beta * (r + \gamma Q_w(s', a') - Q_w(s, a)) * \nabla_w Q_w(s, a)$$

Equation 19: A2C Value Update

A variation of Actor-Critic that uses an advantage function is called Advantage Actor-Critic (A2C). It can significantly reduce the variance of the policy gradient [Silver, 2015]. The advantage function tells us about the average goodness of taking the action in that state [Simonini, 2018].

$$A(s, a) = (\approx Q^{\pi_\theta}(s, a)) - (\approx V^{\pi_\theta}(s))$$

Equation 20: A2C Advantage function

$A(s, a) > 0$, agent is performing above average, gradient is pushed in the same direction

$A(s, a) < 0$, agent is performing worse than the average, gradient is pushed in the opposite direction

As justified in the thesis [Degris et al., 2012], it is possible to execute A2C in an off-policy manner using an experience replay method bringing much efficiency.

2.6 Multi-Agent Deep Reinforcement Learning

The goal of this thesis is to study the multi-agent system using deep reinforcement learning (MADRL). The thesis explores multiple approaches to a multi-agent setup in a stochastic environment. Mainly neural network is used to estimate Q-Values in a stochastic process. Many of the traditional approaches for solving multi-agent systems failed due to the environment is either too complex or the reason that the action space is enormous. This thesis tries to elevate some of the hurdles in multi-agent learning. The thesis mainly follows the work mentioned in [Gupta et al., 2017]. It extends their work to actor-critic algorithm multi-agent Advantage Actor-Critic (MAA2C). It presents the evaluation results as a comparison between actor-critic and two other approaches viz. the vanilla policy gradient Reinforce (MAR) and multi-agent DQN (MADQN) in a multi-agent step. The environment presented here is discrete and uses deep reinforcement learning optimisation techniques to learn better policies. In MADRL, the neural network learns the properties of the multi-agent setup directly from the raw observations and rewards signals [Gupta et al., 2017].

Extensive research has been done in the area of multi-agent setup. [Gupta et al., 2017], cited in [Tampuu et al., 2015] investigated how two agents controlled by independent Deep Q-Networks interact in the classic videogame Pong. They demonstrate that how proper rewards configuration encourages cooperative behaviour among the agents. There are many works in multi-agent learning, nonetheless, very little progress has been done in the area of de-centralised learning involving high dimensional observations and continuous space.

The major challenge around any multi-agent setup is to adapt it to any arbitrary number of agents without changing the underlying architecture. To address this problem, [Stanford, 2016] proposed to make three simplifying assumptions; (i) represent the environment in a two-dimensional format, (ii) discrete action space and (iii) fixed categories of agents. This thesis engraves these assumptions while interacting with the multi-agent setup Pursuit Evasion. More information about the environment is explained in the next section. The thesis chiefly explores the behaviour of agents in a co-operative set up which forces them to make a rational judgement and test agent's cognitive intelligence. Such traits once learned can be helpful in multi-agent systems such as the economy, traffic, and environmental challenges as stated in [Leibo et al., 2017].

Multi-agent reinforcement learning thus boosts collaboration among agents which in turn laid the foundation for tackling many cooperative tasks exists around us.

3. Methodology

3.1 Problem Formulation

The primary goal of this thesis is to evaluate the performance of agents in a co-operative environment under given circumstances. To facilitate this objective, we choose Pursuit-Evasion setup as our evaluation drive as explained in [Vidal cited in [Gupta et al., 2017]. In this setup, there are two sets of agents – Pursuers and Evaders. They both have different objectives. A set of pursuers are venturing to catch the set of evaders around a randomly generated environment. Evaders follow a heuristic policy. They try to stay away as far as possible from evaders. We are interested in building the policy of evaders using MADRL approach which will teach them to catch all the evaders. In this thesis, we have stripped the environment to finite state space. The observation and action spaces are discrete.

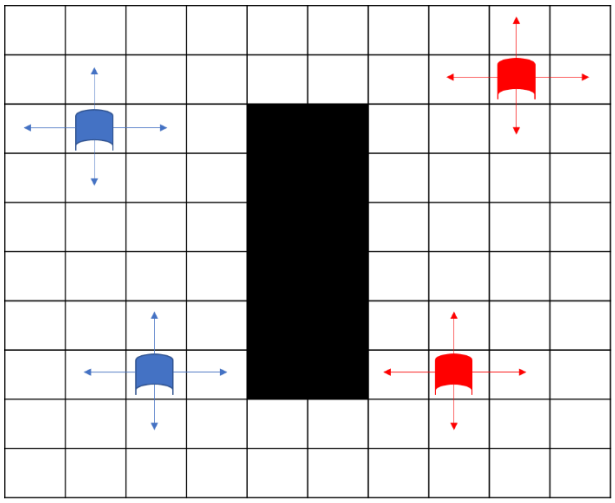


Figure 10: Pursuit-Evasion Environment (for reference only). Two sets of pursuers (Red) are attempting to catch two sets of evaders (Blue) in a 16x16 setup. The black structure at the centre is an obstacle through which agents cannot pass. The arrow indicates the number of cells agents can see at any time.

In this thesis, the environment selected for the experiment is 16x16 with 20 evaders and 10 pursuers. Pursuers have an observation range of 10, meaning they can see up all the cells surrounding them at any time. To catch the evaders, pursers must co-operate to surround an evader from 2 sides. At any time, two pursers must be present surrounding an evader to remove that evader from the play. Pursers get a reward of 5.0 for catching an evader. They also get a reward of 0.01 for sighting an evader without catching them but will ease the exploration process. Each pursuer has five actions and must choose between East, West, South, North or Stay.

As explained in [Stanford, 2016], the observation is represented in an Image like tensor format with 4 channels. The tensor is of the size W x H x 4. W and H are the width and height of our two-dimensional pursuit-evasion domain. 4 indicates the number of channels and each of them encodes distinct information about the environment.

Information Channel: contains information about the environment and the obstacles it contains.

Evaders Channel: contains information about the location of evaders.

Pursers Channel: contains information about the location of pursuers.

Agent Channel: contains information about the agent taking an action.

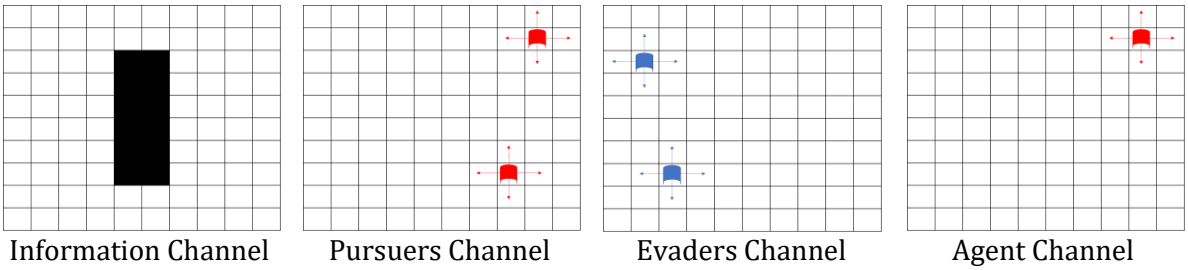


Figure 11: W x H x 4 Image Like Tensor representation of Observation Space in a 10x10 domain.

The major issue with this tensor like input representation is that if the evaders are occupying the same location in the domain, their observational output will be the same. The learning of multi-agent setup will not be fruitful in this case. Hence, the work presented here enforce stochastic learning of policies via taking a softmax over the Q-Values (MA DQN) and softmax over actions probabilities (MA Reinforce or MA A2C). More information on architecture is available in section 3.3.

3.2 MADRL Environment

The learning environment MADRL used in this thesis was created at Stanford Intelligent Systems Laboratory [Gupta et al., 2017]. The MADRL library is fully compatible with the OpenAI Gym platform [Brockman et al., 2016] and provides standard method calls for interacting with the environment. The library is open source and is available on GitHub. It has a dependency on another RL project rllab [Duan et al., 2016] created under MIT license. It is a framework for developing and evaluating reinforcement learning algorithms. The MADRL setup supports python 3.5 and contains an implementation of libraries in Theano as well as TensorFlow. For more specific version information about the libraries, please check the Table 2 of appendices section at the end of this thesis. This rllab toolkit along with MADRL library allows one to develop, train and test multi-agent behaviour using different RL approaches on their multi-agents' environments.

MADRL consist of below major modules:

- MADRL environments contain a multi-agent implementation of various popular and classic environments. This thesis is using Pursuit-Evasion setup.

- Rllab: RL toolkit for developing and evaluating reinforcement learning algorithms.
- Python Application Programming Interface (API): Dedicated Python APIs for various RL functions.

The various MADRL environments are designed while keeping OpenAI Gym function calls at the base. Hence, they provide full compatibility in terms of methods calls and follows basic principles. MADRL also supports parallel processing of tasks in a multi-core CPU. It also has support to evaluate and record a performance in a video or image format.

Setup information:

To setup and train any environment in MADRL, it is necessary to configure below components [Gupta et al., 2017]:

- *Runner class*: One runner for each multi-agent environment. It contains all your environment-specific initialisation parameters. Here, you must define all your setup specific configuration and number of the cores to be used by the CPU. Your multi-agent environment is initialised in this class and is pass to the next component in the chain.

- *Rllab consumer class*: It looks after the creation of a Policy network depending on the network architecture. As of now, it supports Multilayer Perceptron, Convolutional and Recurring configuration.

- *Algorithm class*: It is the first class that takes control of the consumer class. Its job is to create optimiser, initialise every everything, define your loss function and pass the control to the base class.

- *Base Algorithm Class*: Calls constructor and sets everything parameters. Create sampler class to sample experiences of agents. Initialise logger and timer, create multi-core workers and start the training process.

- *Sampler Class*: Returns samples of experiences by acting on the environment. The operation takes place parallelly if multi worker output is on.

- *Base Sampler Class:* It performs aggregation operation on the samples' output by sampler class. The operation is algorithm-specific and can be anything from accumulating rewards to calculating advantage function.

All the information about the environment and configurations including GitHub repository are available in the appendices section.

3.3 Model Architecture

As our observation space is an image-like tensor, we took convolutional layer as per model input. The exact architecture is shown in Figure 12 and is as follows. The input to the convolutional neural network consists of an 16x16x4 image produced by taking a step into the multi-agent environment. The first hidden layer convolves 32 filters of 4x4 with stride 2 with the input image and applies a rectifier nonlinearity. This is followed by a second convolutional layer that convolves 64 filters of 3x3 with stride 1 followed by a rectifier. The final hidden layer is fully connected and consists of 512 rectifier units. The output layer is a fully connected linear layer with a softmax output over the actions. The model presented here used batch-normalisation after every hidden layer.

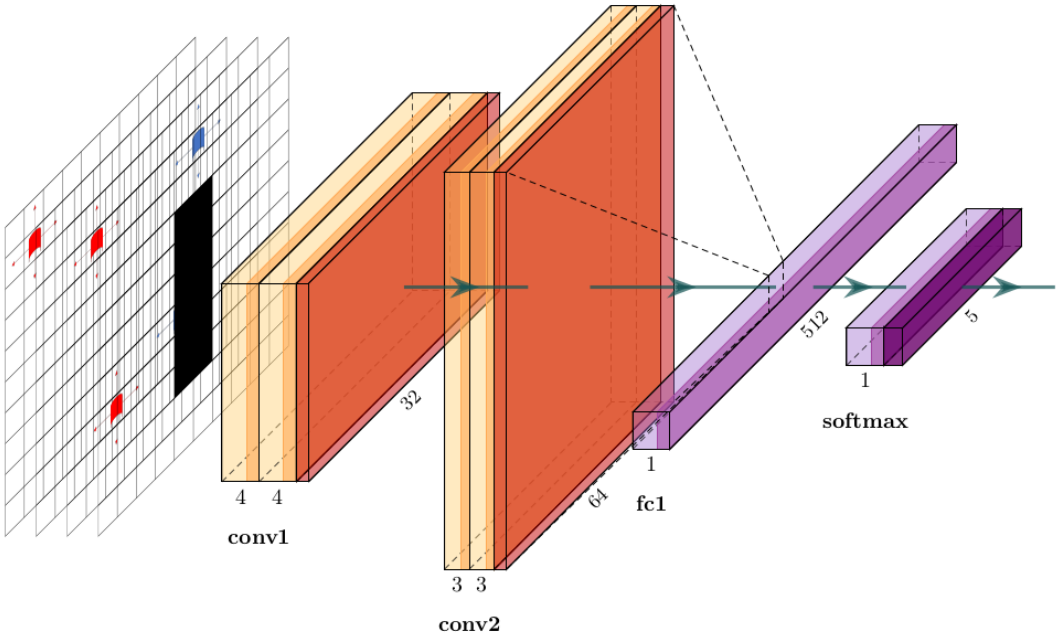


Figure 12: Multi Agent Model Architecture

To train the above network architecture, a dataset containing samples of experiences are generated through simulation. Each sample consist of a tuple of 5 parameters $(o, a_t, r_t, o_{t+1}, d)$. o is the initial image like tensor, a_t is action taken, r_t is a reward for taking current action, o_{t+1} is next observation of the environment and d is the terminal state of the given step. The architecture in Figure 12 is adopted with few modifications from [Mnih et al., 2015]. The softmax layer outputs 5 values that represents the Q-values for all the actions taken in case of MADQN. While it indicates the probability of taking that action in case policy gradient. The architecture presented here uses stochastic policy by taking softmax over the actions. This encourages the exploration process without having to use an epsilon greedy approach. The entropy will go down eventually with the training and the agent will start making exploitation.

3.4 De-centralised Parameter Sharing

As explained in [Lowe et al., 2017], the major problem with any multi-agent RL system is the non-stationarity of the environment. If the policy of each of the agents is different, then the agent may end up learning policy by overfitting to the behaviour of the opponents. Such policies don't last long and become brittle when the opponents alter their strategies.

To obtain a multi-agent policy that is more robust to the changing behaviour of the opponents, this thesis weighs on parameters sharing in a decentralised setup.

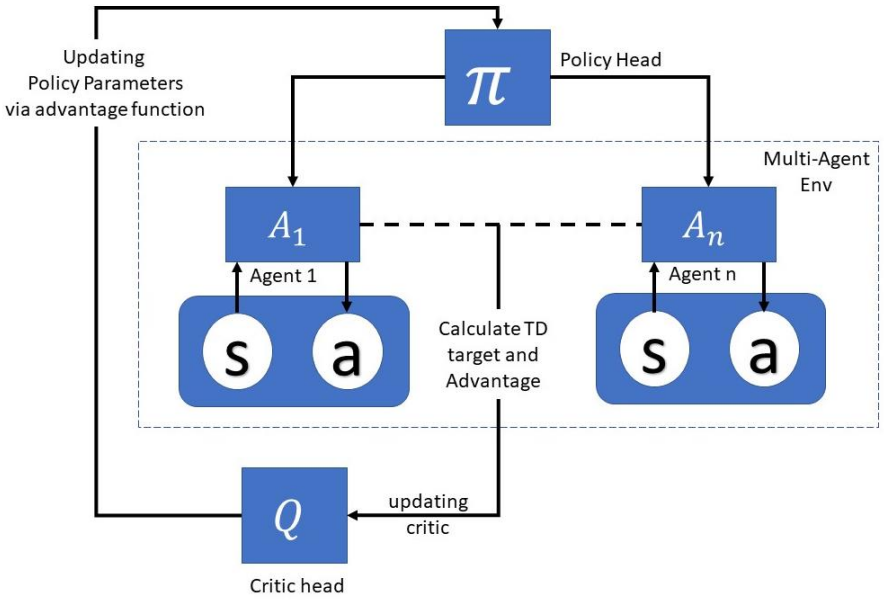


Figure 13: De-centralised control with single policy and centralised critic, s is the observation state, a is the action taken, π is the central policy and Q is the critic function (in case of MAA2C)

In a cooperative task, agents share mutual interests and their paths are uniform. This thesis employs parameter sharing among homogenous agents to encourage this common objective. In parameter sharing, agents share the single central policy so that the learning is centralised. The policy optimisation is carried out via the training experience of all the agents. As shown in Figure 13, the setup is called de-centralised control but centralised learning. This is the most scalable approach as mentioned in [Gupta et al., 2017]. Each agent has a policy network, but they all share the same policy parameters. The agents take different actions since the policy is stochastic and the input observation state is distinct for each agent. Thus, exploration is encouraged. The sampled experiences of all the agents are then aggregated to compute baseline and advantage values. The final step is to maximise the objective function.

For MA A2C, the model has a centralised Critic function. As shown in Figure 13, after combining agents' experiences and calculating advantage values, we optimise the critic network. At the same time, the critic network is used to update the central policy in the direction shown by the critic.

4 RL Algorithms

4.1 Multi-Agent Deep Q-Networks (MADQN)

Algorithm 1 the pseudo code for multi-agent deep Q-networks, adapted from [Mnih et al., 2015].

Input: Image like tensor
Parameter: learning rate $\alpha > 0$
Output: Q action value function

function MADQN
Initialize replay memory D and fill up to size S with random experiences 1
Initialize action-value function Q with random weight θ 2
Initialize target action-value function $\hat{Q}$ with weights θ 3
for episode = 1 to M **do** 4
    for timestep t=1 to max-time-step **or** terminated **do** 5
        for each agent $i = 1$ to N **do concurrently** 6
            initialize s, the first state of the episode 7
            Update policy parameters with central policy 8
            Select $a_t = argmax_a Q(s, a; \theta)$ 9
                Act a_i in emulator and observe reward r_t and state s_{t+1} 10
            Store Transition (s_t, a_t, r_t, s_{t+1}) in D. 11
        end for
        Sample random minibatch of transitions (s_t, a_t, r_t, s_{t+1}) from D 12
        Set $y_i = \begin{cases} r_j, & if\ terminates\ at\ j+1 \\ r_j + \gamma max_{a'} \hat{Q}(s_{j+1}, a'; \theta^-), & otherwise \end{cases}$ 13
        Do GD update on $(y_i - Q(s_j, a_j; \theta))^2$ w.r.t the network θ 14
        Every C step update $\hat{Q} = Q$ 15
    end for
end for
end function

Algorithm 1: MA DQN adapted from [Mnih et al., 2015]

Explanation:

The full algorithm for training multi-agent deep Q-networks is presented Algorithm 1. Input to the algorithm is a 10x10x4 image like tensor as explained in [section 3.1]. Step 1, 2 and 3 initialise network infrastructure. Both Q network and target $\hat{Q}$ consist of a feature layer as convolution. Replay memory is a dequeue that stores agent's experience

(s_t, a_t, r_t, s_{t+1}) at each time-step t in the dataset D. We fill replay memory with experiences of random play with an environment [step 1]. We iterate over M number of episodes [step 4]. Q-Learning is a TD(0) learning and is off-policy. Hence, for each time-step [step 5], we apply Q-learning updates to samples of experience in the inner loop of the algorithm [step 6]. Prior to obtaining every experience, each agents' policy updates its parameters from the central policy [step 8]. Each agent selects and executes action chosen by taking argmax over the softmax output on the Q network [step 9]. Co-operative agents interact with the environment and produce experience tuples which are then aggregated and stored in the replay memory [step 10 and 11]. The algorithm picks samples drawn uniformly at random from the replay memory and passes them to the learning model [step 12]. The learning model calculates expected returns $Q(s_{t+1}, a)$ as TD target using the target network $\hat{Q}$. Use of target network increases the stability of the network against the unwanted oscillations as explained in 0 [step 13]. The above algorithm uses Double Q learning as proposed in [van Hasselt et al., 2015] and 0. The last step is to perform gradient descent on the TD target obtained from the double DQN operation [step 14]. Every C time-step, we update the weights of the target network $\hat{Q}$ from the network Q [step 15].

4.2 Multi Agent Reinforce (MA Reinforce)

Algorithm 2 presents the pseudo code for multi-agent Reinforce with baseline in the episodic case, adapted from [Sutton and Barto, 2018].

Input: policy $\pi(a|s, \theta)$, state value function (baseline) $\hat{v}(s, \omega)$
Parameters: step size, $\alpha > 0, \beta > 0$
Output: policy: $\pi(a|s, \theta)$

function MA REINFORCE

Initialise policy parameter θ and state value (baseline) weights w 1
for episode = 1 to M **do** 2
  for each agent i = 1 to N **do concurrently** 3
    Update policy parameters with central policy 4
    initialize s, the first state of the episode 5
      Select $a_0 = argmax_a Q(s, a; \theta)$ 6
      for time-step t=1 to max-time-step **or** terminated **do** 7
        generate episode $s_0, a_0, r_1, \ldots, s_{T-1}, a_{T-1}, r_T$ following $\pi(.|., \theta)$ 8
      end for
  end for
  $G_T \leftarrow$ *total returns from step t* 9

Calculate advantage value, $\delta = \hat{A}(s_t, a_t) \leftarrow G_t - \hat{v}(s_t, \omega)$ 10

Update baseline parameters using total returns, $\omega \leftarrow \omega + \beta\delta\nabla_\omega \hat{v}(s_t, \omega)$ 11

Update policy parameters using the calculated advantage

$\theta \leftarrow \theta + \alpha\gamma^t \hat{A}(s_t, a_t)\nabla_\theta log\pi(a_t|s_t, \theta)$ 12

end for

end function

Algorithm 2: MA Reinforce adapted from [Sutton and Barto, 2018]

Explanation:

The full algorithm for training multi-agent Reinforce is presented in Algorithm 2. Input to the algorithm is a 10x10x4 image like tensor as explained in [section 3.1]. Here, the parameters are α - learning rate for policy network and β - learning rate for baseline. Step 1 initialise the network infrastructure. Policy network consists of a feature layer as convolution. The output of the policy head is softmax over the number of actions. Baseline act as a linear state value function whose value does not depend on action a. We iterate over M number of episodes [step 2]. For each of the agent, we concurrently act on the environment and get a batch of experiences in the inner loop of the algorithm [step 3]. Prior to obtaining every experience, each agents' policy updates its parameters from the central policy [step 4]. We reset the environment and get the initial state [step 5]. Each agent selects and executes action chosen by taking argmax over the softmax output on the policy network [step 6]. Agent act on the environment for maximum time steps or until the environment is terminated and collect experiences tuples $s_0, a_0, r_1, \ldots, s_{T-1}, a_{T-1}, r_T$ following current policy $\pi(.|.,\theta)$ [step 7 and 8]. The experience samples from all the agents are aggregated and send to the learning model. Reinforce is a Monte Carlo algorithm and is well defined only for the episodic case. Thus, the learning model calculates the total return G_T from time t, which includes all the future rewards up until the end of the episode or max time steps [step 9]. $\hat{v}(s_t, \omega)$ is a linear baseline which outputs the expected returns of following the policy π in the state s. By subtracting $\hat{v}(s_t, \omega)$ from G_T, we get the advantage function $\hat{A}(s_t, a_t)$ [step 10]. Now, we update the baseline using total returns [step 11] and perform a gradient-ascent step on the policy network with advantage value [step 12].

4.3 Multi Agent Advantage Actor Critic (MA A2C)

In actor-critic algorithms, the critic updates action-value function parameters, and the actor updates policy parameters, in the direction suggested by the critic. A2C is a fully online, incremental algorithm. Algorithm 3 presents the pseudo code for the one-step actor-critic algorithm in the episodic case, adapted from [Sutton and Barto, 2018].

Input: policy $\pi(a|s,\theta)$, state-value (critic) $\hat{v}(s,\omega)$
Parameters: step size, $\alpha > 0, \beta > 0$
Output: policy: $\pi(a|s,\theta)$

function MA A2C

Initialise policy parameter θ and state-value (critic) weights w 1
for episode = 1 to M **do**..... 2
    for time step t =1 to max-time-step **or** terminated **do**..... 3
        for each agent i = 1 to N **do concurrently**..... 4
            Update policy parameters with central policy 5
            initialize S, the first state of the episode 6
            Select $a_0 = argmax_a Q(s,a;\theta)$ 7
            Act action a_0 and observes s', r 8
            $\delta \leftarrow r + \gamma\hat{v}(s',\omega) - \hat{v}(s,\omega)$ $(if\ s' is\ terminal, \hat{v}(s',\omega) = 0)$..... 9
            Calculate TD error as advantage $\delta - \hat{v}(s,\omega)$..... 10
            Update critic parameters using the TD target
            $\omega \leftarrow \omega + \beta\delta\nabla_\omega \hat{v}(s_t,\omega)$..... 11
                Update actor parameters using the TD error as advantage
            $\theta \leftarrow \theta + \alpha I\gamma^t \delta\nabla_\theta log\pi(a_t|s_t,\theta)$..... 12
            $a \leftarrow a'$..... 13
            $s \leftarrow s'$..... 14
        end for
    end for
end for
end function

Algorithm 3: MA A2C adapted from [Sutton and Barto, 2018]

Explanation:

The full algorithm for training multi-agent actor-critic is presented in Algorithm 3. Input to the algorithm is a 10x10x4 image like tensor as explained in [section 3.1]. Here, the parameters are α - learning rate for policy network and β - learning rate for critic function. Step 1 initialise the network infrastructure. Policy network and critic both consist of a feature layer as convolution. The output of the policy head is softmax over the number of actions. While the output of the critic head is a single value with linear activation. We iterate over M number of episodes [step 2]. For every time step until the maximum step or the environment state is terminated, agents bootstrap their way in the inner loop of the algorithm [step 3 and 4]. The degree of bootstrapping is configurable. Prior to obtaining every experience, each agents' policy updates its parameters from the central policy [step 5]. We reset the environment and get the initial state [step 6]. Each

agent selects and executes action chosen by taking argmax over the softmax output on the policy network [step 7]. An agent acts on the environment with action a_0 and observes the state and s' and r [step 8]. If s' is the terminal state, TD target is just the reward r received, otherwise it is calculated as the total expected outcome from the state s [step 9]. By subtracting critic estimate $\hat{v}(s_t, \omega)$ from the TD target δ, we get the advantage function $\hat{A}(s_t, a_t)$ [step 10]. The critic estimator function is optimised using stochastic gradient descent on the TD target value [step 11]. Similarly, we perform the gradient-ascent step on the policy network using the advantage value [step 12]. We update the next state and next action [step 13 and 14].

5. Experiments and Results

This segment presents evaluations of the co-operative policy learned by the multi-agent version of DQN, Reinforce and A2C in a Pursuit-Evasion domain. The experiment consists of a 16x16 version of the domain with one long rectangular obstacle (white) in the middle of the environment as shown in Figure 14.

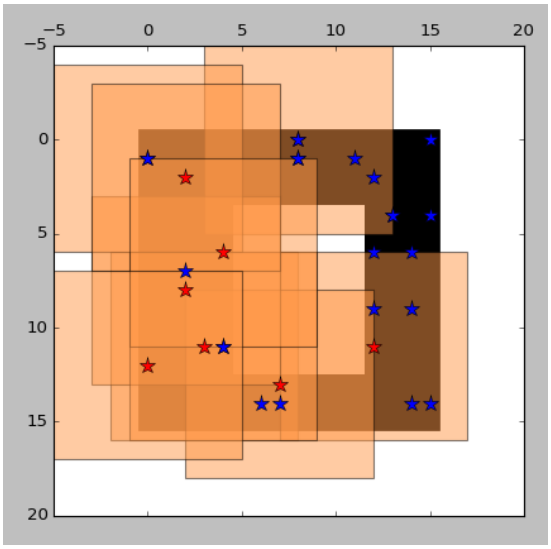


Figure 14: 16x16 Pursuit-Evasion domain with 20 evaders and 10 pursuers.

This thesis compares the capabilities of multi-agent algorithms to adapt to the dynamics of the environment. All the initial hyper-parameters used for evaluation are available in Table 4,Table 5 andTable 6 of the appendices section. The arena as shown in Figure 14 consists of 20 evaders (blue) and 10 pursuers (red). Pursuers can see about 10 blocks surrounding them. To catch the evaders, the pursuers must learn to co-operate and surround the evaders on any two sides. The obstacle in the environment remains fixed and uniform for every run and across all algorithms.

Reward Shaping

There are many benefits of reward shaping. It encourages the agent to explore without worrying about the main objective [Ng et al., as cited in Gupta et al., 2017]. The process involves adding a small bonus reward to the final target that may accelerate the learning process while keeping the quest active and hopeful. The pursuit-evasion environment encourages exploration to pursuers by giving them a small reward of 0.01 for encountering an evader. The decentralised process as described in section 3.4 assigns rewards to agents as per their exploration and intelligence. Contrary to assigning rewards at the global level, only those agents involved in actions around the evaders get a reward. This way of assigning reward is called Local reward signals. It has proved to be useful in reducing the number of samples required for learning [Bagnell and Ng, as cited in Gupta et al., 2017].

Results

The evaluation presented here is based on the two important properties: the total reward and the average reward. The total-reward gives insight into how far the exploration has grown so far. It is the sum of rewards of all the agents per iteration. The average reward tells us how well the model can handle all the states together. The training uses a batch size of 500 for MA reinforce and MA A2C with a maximum step size of 12000. In the case of MA DQN, the same model architecture is used for the policy and target network. The batch is size kept at 32 with a maximum step size of 200. The learning rate for the policy network is 1e-7 and for the q network is 1e-3.

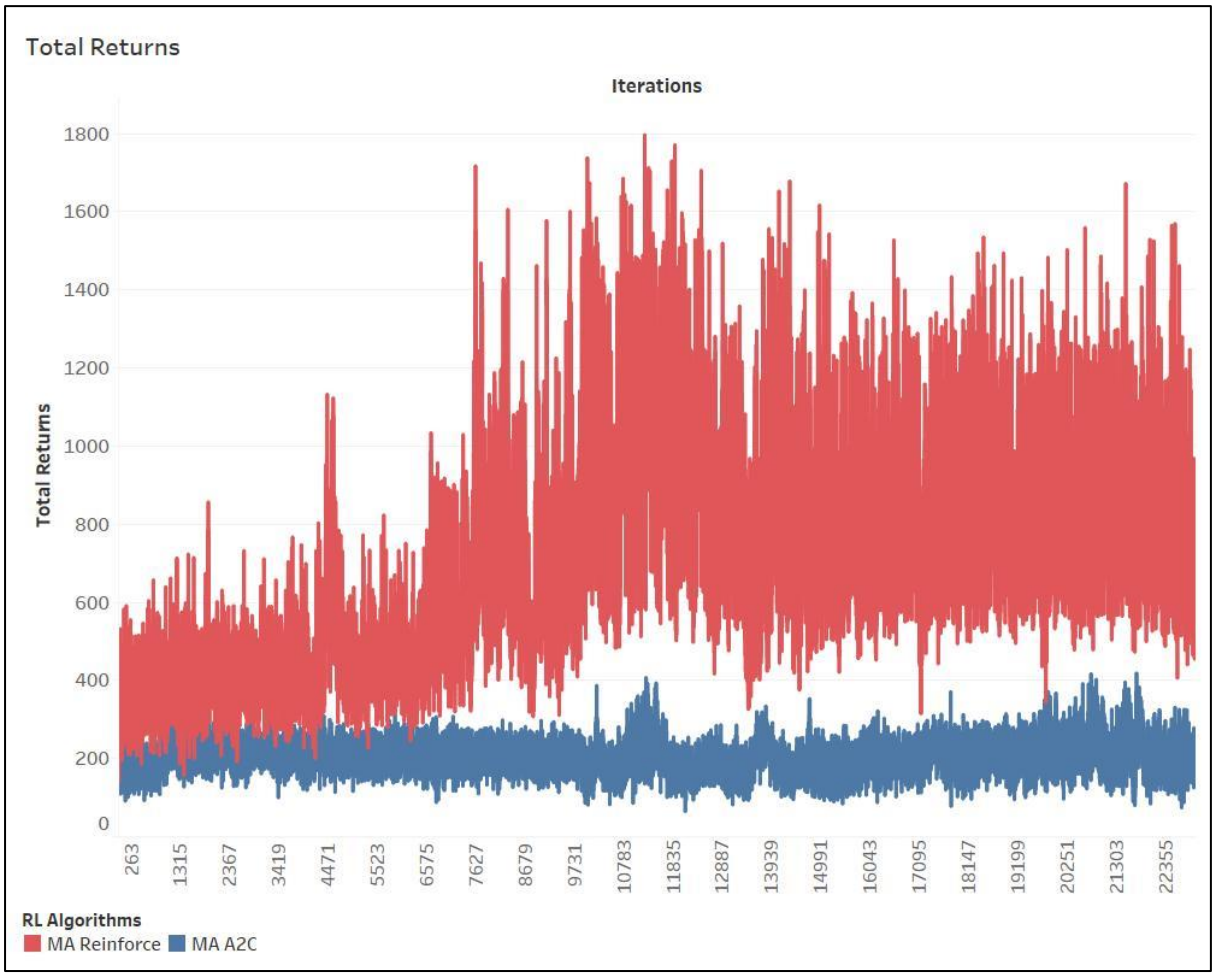


Figure 15: Total Rewards for multi agent algorithms during the training run.

We see that the MA Reinforce performed well than the MA A2C in a given setting. After Iteration 6000, there is a small rise in the total rewards for Reinforce. This is the point where the model started to learn to break up pursuers into a team of two to catch evaders. As total rewards account for rewards gained by all the agents per iteration, the significant increase in later iterations shows that MA Reinforce was able to adapt to the situation quickly. This also verifies the point that MA Reinforce was able to understand the spatial correlation of the domain using the convolutional neural network architecture. MA A2C had quite a few ups and downs around the iteration 11000. We also see an upward surge

around the iteration 21000. The reason for such oscillations might be that MA A2C is still in the learning phase and performance may get improve with more training. More information on this behaviour is discussed in the entropy section. Figure 15 doesn't say anything about the consistency in the learning process across training runs. We try to uncover this in the next section where we study the distribution of total rewards in MA Reinforce and MA A2C.

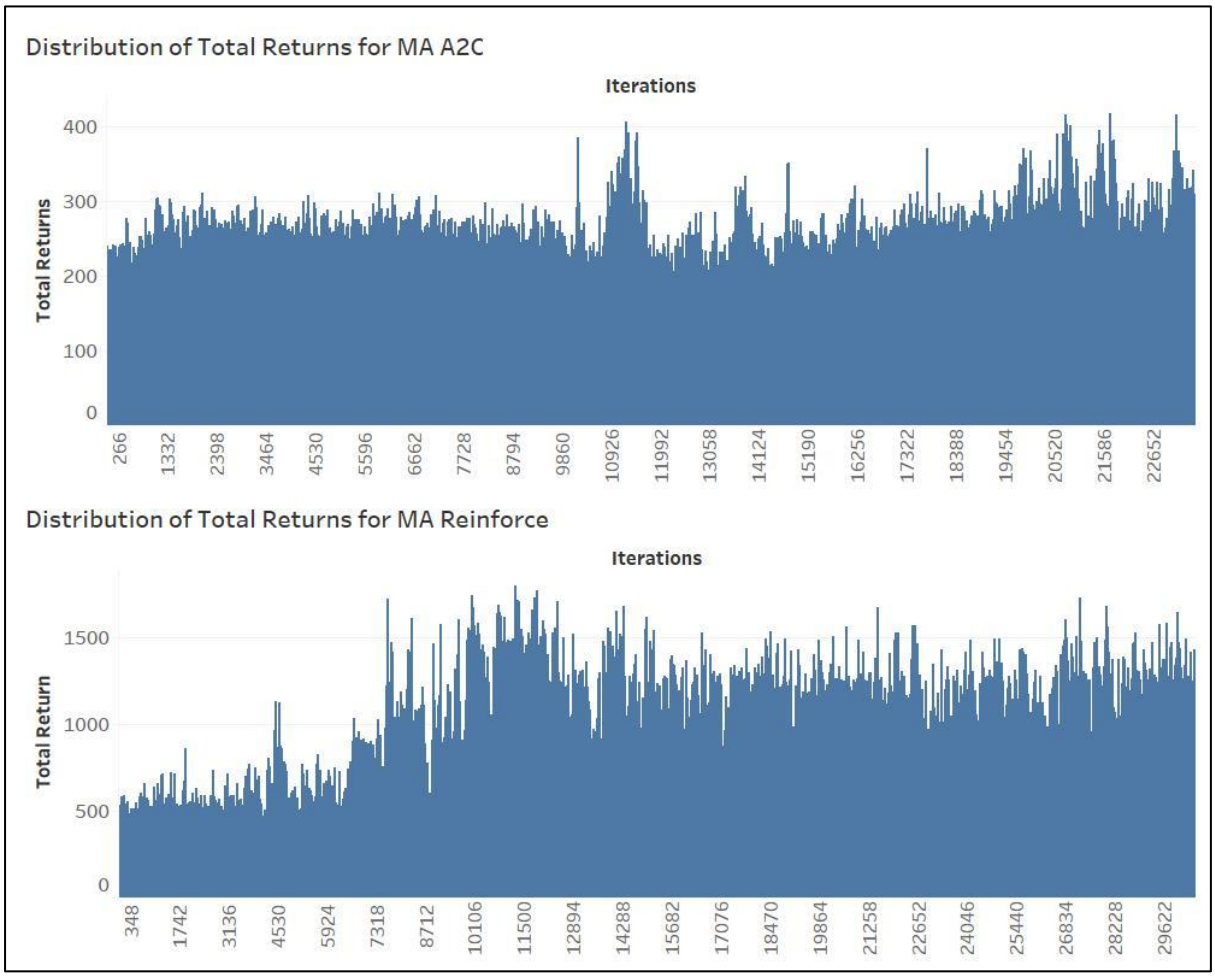


Figure 16: Distribution of Total Returns for MA Reinforce and MA A2C

Figure 16 clearly states that the performance spikes in case of MA Reinforce in later iterations were indeed learned values and not some random hits by the model. We can see that the model gradually determines the policy and the frequency of high rewards goes up. The agents were able to segregate tasks among themselves with more training runs. In the case of MA Reinforce, once the agents formed the group of 2 to surround an evader, they seem to be carrying forward the strategy effectively. For MA A2C, the distribution of Total Reward is substantially uniform across iterations. Except for the few highs in the middle and towards the end, the performance stays around 300 throughout. We see that the model doesn't gradually get better and learns the same values for successive runs.

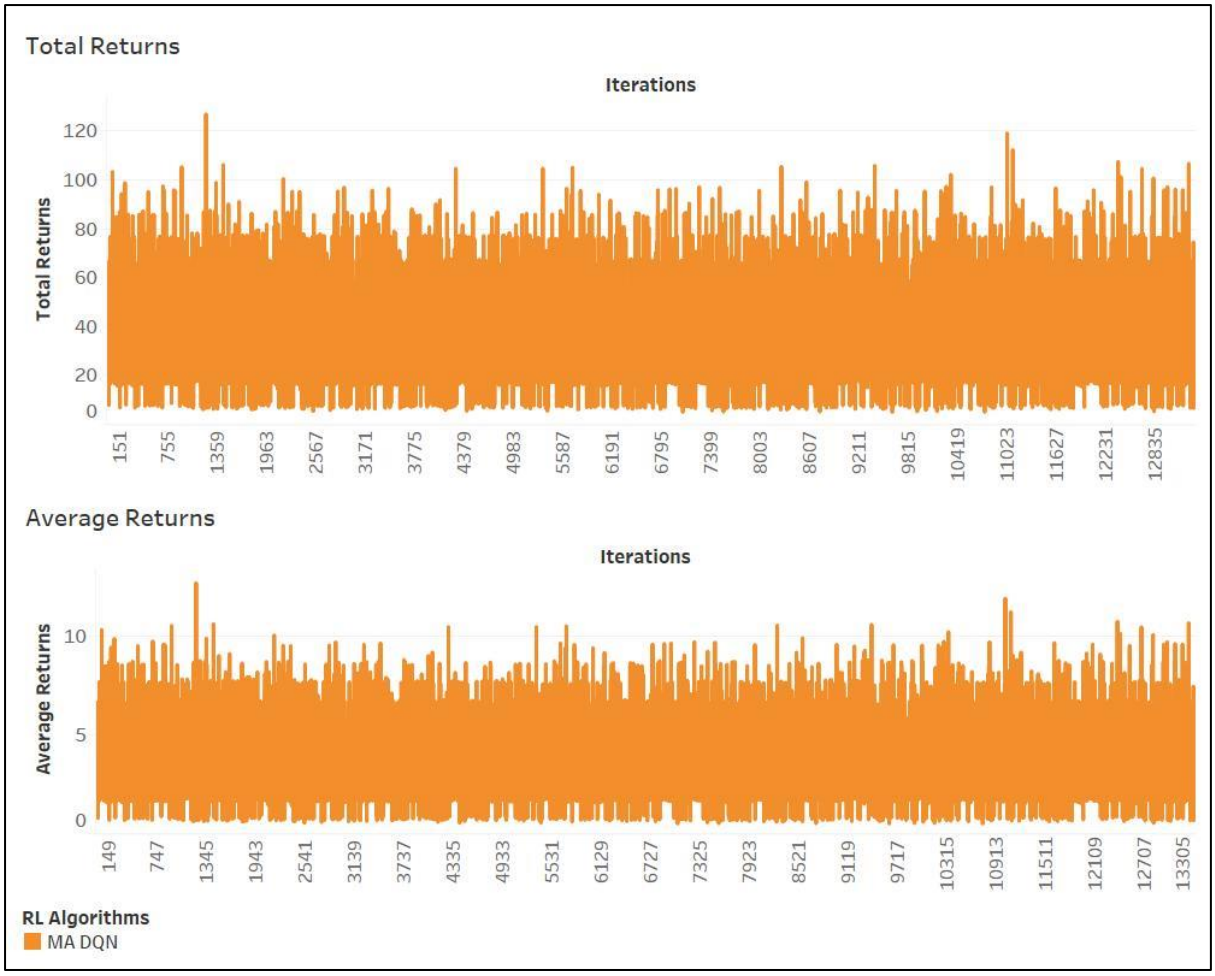


Figure 17: Total Rewards and Average Rewards for MA DQN

MA DQN does not perform as well as the policy gradient-based methods. The max average reward hovers around the value of 10 while the max total reward remains around 100. This shows DQN's inability to learn de-centralised policy in a non-stationary environment. The policy of the agents constantly changes during the training process and thus impact the learning process adversely. As explained in [Gupta et al., 2017], the reason for DQN's inability to learn in a non-stationary environment is due to the quick expiration of the replay memory. The moving dynamics of the domain quickly obsolete the replay memory samples. The results in Figure 17 firmly backed the hypothesis presented above.

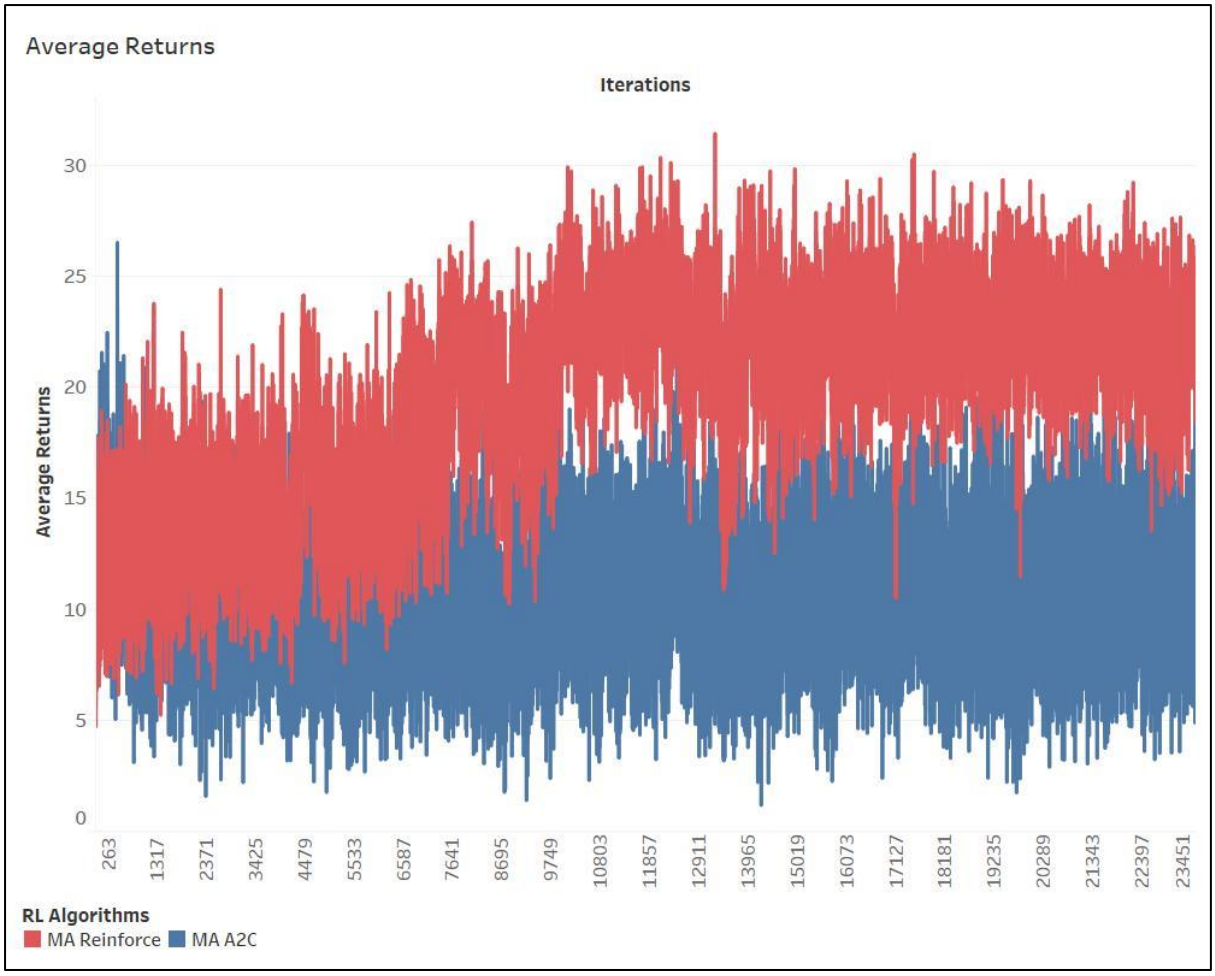


Figure 18: Average Returns

The story depicted by average returns in Figure 18 is not clear enough to explain the growth of our model. Partly because the algorithm keeps on trying new actions as a part of the exploration process. As a result, some actions fail while some hits high resulting in constant oscillations. Nevertheless, there is a modest rise in average returns for MA Reinforce with more iterations. While there is no growth in the case of MA A2C.

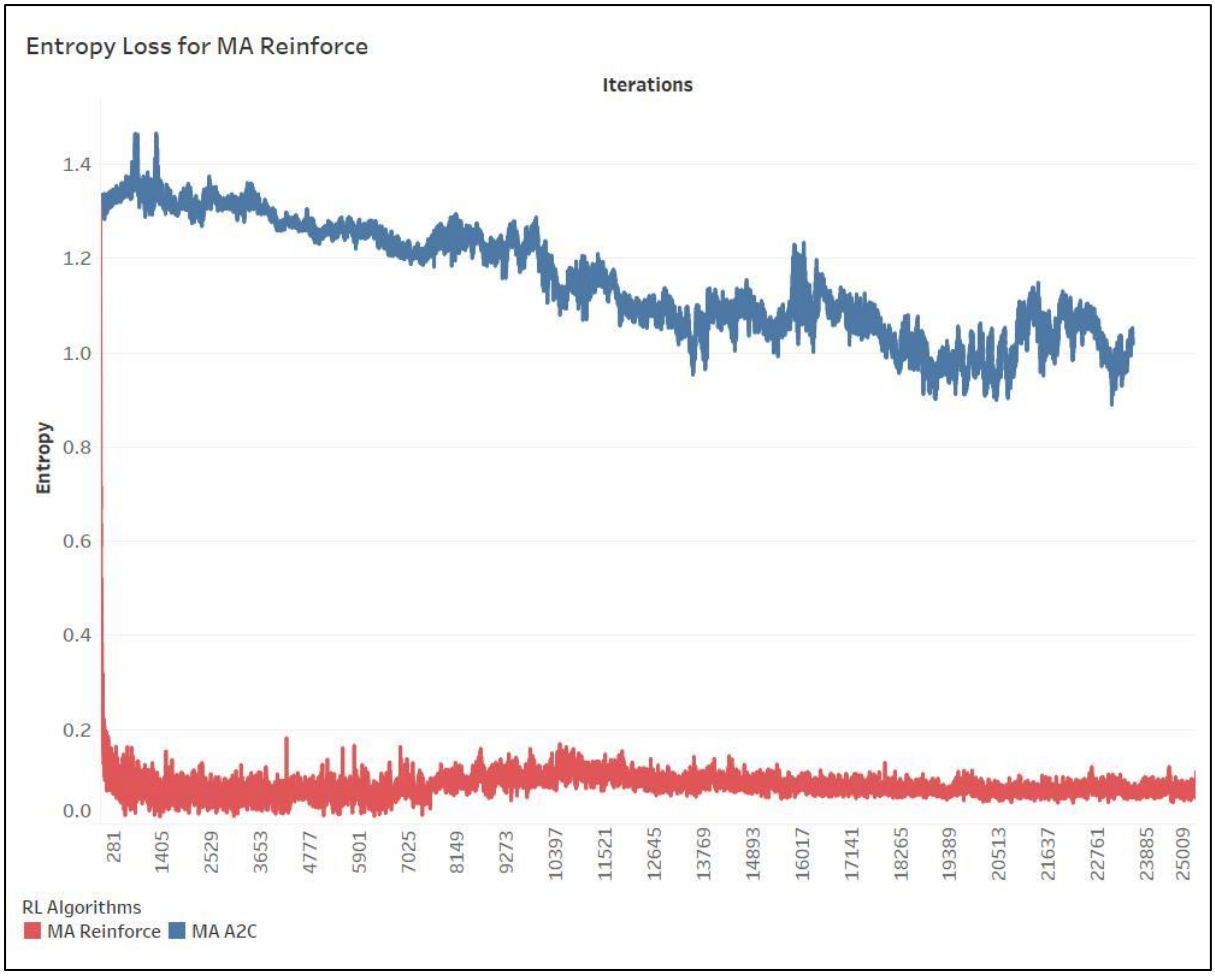


Figure 19: Entropy Loss for MA Reinforce and MA A2C

The entropy graph provides some valuable information about how well the model is performing. For 5 actions, as per the formula, the entropy is 1.61. The model does not perform well as long as the entropy is equal to or nearby 1.61. In Figure 19, we can see that the entropy for MA Reinforce starts at 1.3 and then suddenly drops to 0.2 before going stable for the remaining time. With a significant drop after Iteration 12000, we start to get to high scoring rewards. For MA A2C, entropy starts around 1.3 and then with few hiccups lowers down to 0.9. However, the final entropy is still large which indicates the high probability of a few inaccurate actions. The model needs to learn to bring down the probability of such unwanted actions and give a push to the right ones.

6. Conclusion and Future work

Co-operative Learning has emerged as an innovative approach to tackle various social interdependencies in a multi-participant setup. It is not limited to just sharing understanding and helping each other. It has emerged towards building insight into the domain it is exposed to. Such ground-breaking developments are continually occurring in the field of co-operative learning and knowledge sharing still, multi-agent systems pose a significant challenge to the machine learning process. The biggest obstacle lies in the scalability of the approach to the larger and continuous domain.

In this thesis, we extended three RL algorithms to the multi-agent system using deep learning framework. We selected Pursuit-Evasion environment to test our approaches. The method consists of a convolution neural network that extracts features from an image like tensor input. The model used a stochastic policy to encourage exploration. To share knowledge and boost co-operation, the method employs de-centralised parameter sharing among agents.

This thesis tried to learn the policy in a discrete space. But still, many optimisations are required to learn policies in continuous spaces. Experimental results discussed in this thesis confirm that policy-based methods are better at learning domain non-stationarity than q-learning methods. De-centralized parameter sharing is scalable to a greater number of agents. It allows agents to act on different states per time step using the central policy. This approach expedites the learning and exploration at the same time.

We did not explore the parameter space for deep learning network used in this study. Finding the best configuration of hyperparameters is a daunting task and can be taken care of using methods like grid search. [Gozzoli, 2018]. The distinguished nature of deep learning models makes the optimisation problem very challenging. Additionally, due to time constraints and limited hardware, few training sessions were performed. This entails the possibility to extend the training further possibly with different types of domain scenarios.

Ultimately, it would possible for an agent to converge towards the global maximum via effective training and tuning of hyperparameters in the best environment setting.

References

[Bellman, 1954] Bellman, R. (1954). The theory of dynamic programming.

[Bellman, 1956] Bellman, R. E. (1956). Dynamic programming and lagrange multipliers. *Proceedings of the National Academy of Sciences of the United States of America*, 42 10:767–9.

[Brockman et al., 2016] Brockman, G., Cheung, V., Pettersson, L., Schneider, J., Schulman, J., Tang, J., and Zaremba, W. (2016). Openai gym. *CoRR*, abs/1606.01540.

[Degris et al., 2012] Degris, T., White, M., and Sutton, R. S. (2012). Off-policy actor-critic. *CoRR*, abs/1205.4839.

[Dosovitskiy et al., 2017] Dosovitskiy, A., Ros, G., Codevilla, F., Lopez, A., and Koltun, V. (2017). CARLA: An open urban driving simulator. In *Proceedings of the 1st Annual Conference on Robot Learning*, pages 1–16.

[Duan et al., 2016] Duan, Y., Chen, X., Houthooft, R., Schulman, J., and Abbeel, P. (2016). Benchmarking deep reinforcement learning for continuous control. *CoRR*, abs/1604.06778.

[Ecoffet, 2018] Ecoffet, A. L. (2018). An intuitive explanation of policy gradient.

[Evans and Gao, 2016] Evans, R. and Gao, J. (2016). Deepmind ai reduces google data centre cooling bill by 40%.

[Finnman and Winberg, 2016] Finnman, P. and Winberg, M. (2016). Deep reinforcement learning compared with q-table learning applied to backgammon. Thesis, KTH ROYAL INSTITUTE OF TECHNOLOGY, STOCKHOLM, SWEDEN 2016.

[Gozzoli, 2018] Gozzoli, A. (2018). Practical guide to hyperparameters optimization for deep learning models.

[Gupta et al., 2017] Gupta, J. K., Egorov, M., and Kochenderfer, M. (2017). Cooperative multi-agent control using deep reinforcement learning. In *International Conference on Autonomous Agents and Multiagent Systems*, pages 66–83. Springer.

[Juliani et al., 2018] Juliani, A., Berges, V., Vckay, E., Gao, Y., Henry, H., Mattar, M., and Lange, D. (2018). Unity: A general platform for intelligent agents. *CoRR*, abs/1809.02627.

[Leibo et al., 2017] Leibo, J. Z., Zambaldi, V. F., Lanctot, M., Marecki, J., and Graepel, T. (2017). Multi-agent reinforcement learning in sequential social dilemmas. *CoRR*, abs/1702.03037.

[Levine et al., 2015] Levine, S., Finn, C., Darrell, T., and Abbeel, P. (2015). End-to-end training of deep visuomotor policies. *CoRR*, abs/1504.00702.

[Lowe et al., 2017] Lowe, R., Wu, Y., Tamar, A., Harb, J., Abbeel, P., and Mordatch, I. (2017). Multi-agent actor-critic for mixed cooperative-competitive environments. *CoRR*, abs/1706.02275.

[Mnih et al., 2016] Mnih, V., Badia, A. P., Mirza, M., Graves, A., Lillicrap, T. P., Harley, T., Silver, D., and Kavukcuoglu, K. (2016). Asynchronous methods for deep reinforcement learning. *CoRR*, abs/1602.01783.

[Mnih et al., 2015] Mnih, V., Kavukcuoglu, K., Silver, D., Rusu, A. A., Veness, J., Bellemare, M. G., Graves, A., Riedmiller, M. A., Fidjeland, A., Ostrovski, G., Petersen, S., Beattie, C., Sadik, A., Antonoglou, I., King, H., Kumaran, D., Wierstra, D., Legg, S., and Hassabis, D. (2015). Human-level control through deep reinforcement learning. *Nature*, 518:529–533.

[Silver, 2015] Silver, D. (2015). Reinforcement learning.

[Silver et al., 2016] Silver, D., Huang, A., Maddison, C., Guez, A., Sifre, L., van den Driessche, G., Schrittwieser, J., Antonoglou, I., Panneershelvam, V., Lanctot, M., Dieleman, S., Grewe, D., Nham, J., Kalchbrenner, N., Sutskever, I., Lillicrap, T., Leach, M., Kavukcuoglu, K., Graepel, T., and Hassabis, D. (2016). Mastering the game of go with deep neural networks and tree search. *Nature*, 529:484–489.

[Silver et al., 2017a] Silver, D., Hubert, T., Schrittwieser, J., Antonoglou, I., Lai, M., Guez, A., Lanctot, M., Sifre, L., Kumaran, D., Graepel, T., Lillicrap, T. P., Simonyan, K., and Hassabis, D. (2017a). Mastering chess and shogi by self-play with a general reinforcement learning algorithm. *CoRR*, abs/1712.01815.

[Silver et al., 2017b] Silver, D., Schrittwieser, J., Simonyan, K., Antonoglou, I., Huang, A., Guez, A., Hubert, T., Baker, L. R., Lai, M., Bolton, A., Chen, Y., Lillicrap, T. P., Hui, F. F. C., Sifre, L., van den Driessche, G., Graepel, T., and Hassabis, D. (2017b). Mastering the game of go without human knowledge. *Nature*, 550:354–359.

[Simonini, 2018] Simonini, T. (2018). Improvements in deep q learning: Dueling double dqn, prioritized experience replay, and fixed.

[Stanford, 2016] Stanford, M. E. (2016). Multi-agent deep reinforcement learning.

[Sutton and Barto, 1988] Sutton, R. S. and Barto, A. G. (1988). Reinforcement learning: An introduction. *IEEE Transactions on Neural Networks*, 16:285–286.

[Sutton and Barto, 2018] Sutton, R. S. and Barto, A. G. (2018). *Reinforcement Learning: An Introduction, Second Edition*. MIT Press, Cambridge, MA, 2018, second edition.

[Tampuu et al., 2015] Tampuu, A., Matiisen, T., Kodelja, D., Kuzovkin, I., Korjus, K., Aru, J., Aru, J., and Vicente, R. (2015). Multiagent cooperation and competition with deep reinforcement learning. *CoRR*, abs/1511.08779.

[van Hasselt et al., 2015] van Hasselt, H., Guez, A., and Silver, D. (2015). Deep reinforcement learning with double q-learning. *CoRR*, abs/1509.06461.

[Vidhate and Kulkarni, 2012] Vidhate, D. D. and Kulkarni, P. (2012). Cooperative machine learning with information fusion for dynamic decision making in diagnostic applications. *2012 International Conference on Advances in Mobile Network, Communication and Its Applications*, pages 70–74.

[Wagner et al., 2018] Wagner, J., Baur, T., Zhang, Y., Valstar, M. F., Schuller, B. W., and André, E. (2018). Applying cooperative machine learning to speed up the annotation of social signals in large multi-modal corpora. *CoRR*, abs/1802.02565.

Appendices

Table 1: Hardware Infrastructure

Processor	Desktop variant of the 8th Generation 6-core Intel Core i5-8400 processor
Ram	8 GB
Operating System	Virtual Instance of Ubuntu 18.04.3 LTS
Tensor Type	CPU

Table 2: Libraries and tools

Python	3.5.2
Tensor flow	0.10.0rc0
Keras	2.2.4
NumPy	1.10.4
PyCharm Community	2019.2
Tableau Professional	2019.2.2 64bit

Table 3: Code directory and location

Home directory	https://github.com/dhavalsalwala/rl-algos
MA DQN	**Agent**: https://github.com/dhavalsalwala/rl-algos/blob/master/ma_agents/dqn_agent.py **Runner**: https://github.com/dhavalsalwala/rl-algos/tree/master/rltechniques/multi_agent/dqn
MA Reinforce and MA A2C	**Agent**: https://github.com/dhavalsalwala/rl-algos/blob/master/ma_agents/reinforce_agent.py, https://github.com/dhavalsalwala/rl-algos/blob/master/ma_agents/a2c_agent.py **Runner**: https://github.com/dhavalsalwala/rl-algos/tree/master/rltechniques/multi_agent/policy_gradient
Logs and Results	https://github.com/dhavalsalwala/rl-algos/tree/master/rltechniques/multi_agent/results

Table 4: MA DQN hyperparameters

Hyperparameters	Value	Descriptions
Max episodes	15000	The maximum no. of simulation generated
Max time steps	200	The maximum no. of time steps to follow in an episode
Mini batch size	32	No. of training samples per update
Replay memory size	500000	Size of experience replay dataset
Replay memory pre-trained size	10000	No. of pre-trained experiences in the dataset
Learning rate	1e-3	Learning rate used by the optimiser
Update rule	ADAM	The parameter update rule used by the optimiser
Target network update	2000	Frequency of updating the target network in time steps.
Hidden non-linearity	rectified linear unit (ReLU)	Activation function in hidden layer
Convolutional units	32, 64	Number of conv neural network parameters
Convolutional filter size	4x4, 3x3	Conv filer size
Convolutional strides	2, 1	Convolving strides for filter
Fully connected units	512	No. of neurons in the fully connected layer
Output non-linearity	Softmax	Activation function in the output layer
Batch Normalisation	True	Normalise weights after every hidden layer

Table 5: MA Reinforce hyperparameters

Hyperparameters	Value	Descriptions
Max episodes	25000	The maximum no. of simulation generated
Max time steps	12000	The maximum no. of time steps per episode
Max path length	400	The maximum no. of time steps per batch
Policy network Learning rate	1e-4	Learning rate used by the optimiser
Update rule	Adam	The parameter update rule used by the optimiser
Baseline type	Linear	Type of baseline to calculate advantage function
Hidden non-linearity	rectified linear unit (ReLU)	Activation function in the hidden layer
Convolutional units	32, 64	Number of filters
Convolutional filter size	4x4, 3x3	Convolution filer size
Convolutional strides	2, 1	Convolving strides for filter
Fully connected units	512	No. of neurons in the fully connected layer
Output non-linearity	Softmax	Activation function in the output layer
Batch Normalisation	True	Normalise weights after every hidden layer

Table 6: MA A2C hyperparameters

Hyperparameters	Value	Descriptions
Max episodes	25000	The maximum no. of simulation generated
Max time steps	12000	The maximum no. of time steps per episode
Bootstrapping degree	400	The maximum no. of time steps per batch
Policy network Learning rate	1e-7	Learning rate used by the optimiser
Update rule	Rmsprop	The parameter update rule used by the optimiser
Critic head	Value estimator	Critic estimator to calculate TD target
Hidden non-linearity	rectified linear unit (ReLU)	Activation function in the hidden layer
Convolutional units	32, 64	Number of filters
Convolutional filter size	4x4, 3x3	Convolution filer size
Convolutional strides	2, 1	Convolving strides for filter
Fully connected units	512	No. of neurons in the fully connected layer
Output non-linearity	Softmax	Activation function in the output layer
Batch Normalisation	True	Normalise weights after every hidden layer