Pursuit Evasion Using Deep Reinforcement Learning and Messaging

Rohit R R and **Shalabh Bhatnagar**
Electrical Engineering Department, IISc
Computer Science and Automation Department, IISc
Indian Institute of Science
Bangalore 560012, India

Abstract

Pursuit Evasion is a problem where one or more pursuers try to capture one or more evaders. This problem has been a topic of research in optimal control, game theory and reinforcement learning with various techniques used on it from each field. In this paper we try to solve this problem with the help of the recent advancements in Deep Reinforcement Learning. We solve the single agent pursuer evader case with the help of the Deep Recurrent Q-Learning and Deep Q-learning with Priority Replay. Then we propose an architecture to handle the multi-agent case of pursuit evasion where we enable the pursuers to communicate with each other to capture the evader and introduce communication trade-off weights to avoid any biases between communication and the performance. Our algorithm enables the pursuers to capture the evaders effectively by balancing the learning of co-ordination policies and their own individual policies.

Introduction

Pursuit evasion games are games where there are single or multiple pursuers which try to catch single or multiple evaders. Because of its extensive applications, such as searching buildings for intruders, traffic control, military strategy and surgical operation pursuit evasion has been a topic of research in fields of game theory, optimal control and reinforcement learning. Reinforcement learning is usually applied by taking the pursuer to be the agent and making it learn the appropriate policies to capture the evader.

In literature this problem has mostly been dealt in only low dimensional state spaces or by restricting the motions of the pursuers. But practical cases of pursuit evasion need modeling with large state spaces and high levels of control and coordination to obtain adequate results. With the recent advancements in deep reinforcement learning, our idea was to tackle these harder challenges in the pursuer evader game. We refer a single pursuer trying to capture an evader as single-agent pursuit evasion and multiple pursuers chasing a single evader as multi-agent pursuit evasion.

In this paper we first deal with the single-agent pursuit evasion problem by characterizing it as a POMDP. We apply the Deep Recurrent Q-Learning (Hausknecht and Stone 2015) and the Deep Q-Learning with Priority Replay (Schaul et al. 2015) in our algorithm to achieve a learned pursuer. For the multi-agent pursuit evasion, we allow the agents to communicate with each similar to the communication in (Foerster et al. 2016). But in order to remove unwanted biases in learning we use communication trade-off weights to enable accelerated training of the agents. We use this along with the algorithms adopted in the single-agent case to obtain coordinating efficient pursuers.

Background

Reinforcement Learning (RL): We briefly discuss the RL setting that was used. An agent interacts with an environment E over a number of discrete time steps. At each time step t, the agent observes a state s_t and selects an action a_t from a set of possible actions A according to its policy π, where π is a mapping from states s_t to actions a_t . In return, the environment moves to the next state s_{t+1} and the agent receives a scalar reward r_t . The total accumulated return $R_t = \sum_{k=0}^{T} \gamma^k r_{t+k}$ with discount factor $\gamma \in (0, 1)$, is generally maximized by each agent in order to learn the task.

The action value $Q^\pi(s, a) = \mathbb{E}[R_t|s_t = s, a]$ is the expected return for selecting action a in state s and following policy π. The Q-learning algorithm (Watkins and Dayan 1992) tries to find the optimal Q-value function over all the policies.

This setting is for the case when the game is fully observable, that is the agent can observe the full environment states on which the MDP is based. If the environment states are not directly accessible by the agent and instead they only receive some observations O_t from the environment, then the problem becomes a POMDP (Partially observable Markov Decision Process). In which case all these terms will be similarly redefined based on the observations O_t.

Deep Q-Learning: In this algorithm a deep neural network (Mnih et al. 2015) is used as a function approximator for the Q-Learning algorithm. But in order to tackle the issue of instability, two methods - experience replay and target Q-network are used.

The approximate action value function $Q(s, a; \theta)$ with parameters θ represented with a Deep Neural Network(DNN). In one-step Deep Q-learning, the parameters θ of the action value function $Q(s, a; \theta)$ are learned by iteratively minimiz-

ing a sequence of loss functions, where the i^{th} loss function defined as

$$L_i(\theta_i) = \mathbb{E}\Big[\big(r + \gamma \max_{a'} Q(s', a'; \theta_i^-) - Q(s, a; \theta_i)\big)^2\Big] \quad (1)$$

where s' is the state encountered after state s. We update the action value $Q(s, a)$ towards the one-step return $r + \gamma \max_a Q(s, a; \theta^-)$. On differentiation we get the gradient with respect to the parameters θ_i, $\nabla_{\theta_i} L_i(\theta_i) =$

$$\mathbb{E}\Big[\big(r + \gamma \max_{a'} Q(s', a'; \theta_i^-) - Q(s, a; \theta_i)\big)\nabla_{\theta_i} Q(s, a; \theta_i)\Big] \quad (2)$$

which is then used for the stochastic gradient update of the network parameters. θ^- represents the parameters of the target network.

Multi-Agent Reinforcement Learning : A stochastic game is represented by a tuple $< S, U_1, U_2, ..., U_n, f, \rho_1, \rho_2, ..., \rho_n >$ where n is the number of agents, S is the set of environment states, yielding the joint action set $U = U_1 \times U_2.... \times U_n$, $f : S \times U \times S \to [0, 1]$ is the state transition probability function, and $\rho_i : S \times U \times S \to \mathbb{R}, i = 1, 2, 3...., n$ are the reward functions of the agents. In the multi-agent case the state transitions are a result of the joint actions taken by all the agents, $u = [u_1, u_2, ..., u_n]$ where $u_i \in U_i$. The policy functions $h_i : S \times U_i \to [0, 1]$ form together the joint policy h. The Q-function, $Q_i^h : S \times U \to \mathbb{R}$ of each agent i for the multi-agent case depends on the policies of all the agents,

$$Q_i^h(s, u) = \mathbb{E}[\sum_k \gamma^k r_{i,k} | s_0 = s, h] \quad (3)$$

In literature (Busoniu, Babuska, and De Schutter 2008) most algorithms try to solve for the optimal $Q_i^h(s, u)$ over the joint policies. Many multi-agent reinforcement learning algorithms like policy hill climbing (Bowling and Veloso 2001), hysteric Q-learning (Matignon, Laurent, and Le Fort-Piat 2007), joint action learners (Kapetanakis and Kudenko), Nash Q-learning and more (Busoniu, Babuska, and De Schutter 2008) have been proposed all having limitations with non-convergence or slow convergence or scaling issues. In fully cooperative stochastic games, all the reward functions are usually same. Since our pursuit evasion games fall under this category we did not consider the approaches to mixed or non-cooperative stochastic games.

In the case of centralized execution MDPs/POMDPs are solved over the entire joint actions and joint policies. In the decentralized execution scenario each agent follows a reinforcement learning algorithm to solve individual learning problems. Here each agent tries to solve separate MDPs/POMDPs based on mainly their own policy. But in such cases the environment for each agent ceases to be Markovian as the environment state change is not only dependent on the agent's action but also the actions of the other agents. Hence it has been difficult to get good convergent multi-agent reinforcement learning algorithms (Busoniu, Babuska, and De Schutter 2008). But we can use the methods in deep reinforcement learning to overcome these difficulties. Primary focus was on the decentralized learning case since it is more challenging and is advantageous practically in terms of computation and speed.

Construction of the simulator and the Game

The simulator for the pursuit evasion games was built with python. We used mainly PYMUNK and PYGAME python libraries that provide the physical ecosystem which would handle the movements of the objects such that they obey Newtons Laws of Motion. The pursuers and the evader are defined as circular objects with properties of mass and size. Their direction of motion and speed are controlled by accessing their screen coordinates. We add constraints on their movement. Every time step, the screen with the objects in their new location is rendered to view the game.

The green and yellow objects in Figure 1 represent the pursuers and the orange object represents the evader. The white dashed lanes are the sensor fields which detects the evader's presence for the pursuer. The pursuers are controlled by the policies generated by our reinforcement learning algorithms while the evaders have a predefined way of moving.

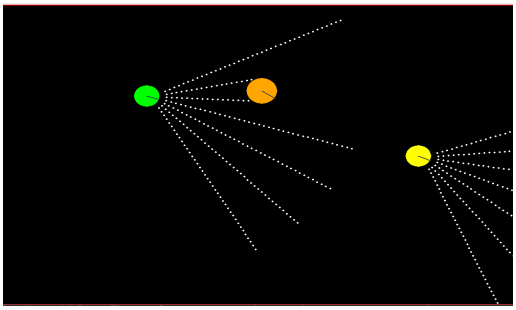

Figure 1: Screen shot of the simulator

Single-Agent Pursuer Evader

In the single agent (pursuer) and evader case, where there is a single pursuer chasing a single evader, we initially started out with a random moving evader. For this case when the evader is moving slowly and randomly, we observed that even though the state space was high a simpler linear function approximator with Q-learning was able to learn quite well to catch the evader. This was because it was sufficient to learn a simple policy even with the large state space. But when the game was modified such that the evader can evade the pursuer in a complex path, the simple Q-learning with linear function approximator no longer was able to learn the task as it was much more complicated than the simple random evader case. In this case of evader following it's own predefined evasion policy, the game gets considerably sophisticated making it suitable for the application of deep reinforcement learning algorithms. The following describes how the main characteristics of the game are defined.

States: The states/observations accessed by the pursuer are the sensor values. Each pursuer might have upto 15 sensors, where each sensor has a range of upto 100.

Actions: The actions of the pursuer are basically to rotate

by a certain degree (between 0.2 and 1.57 radians) clockwise or anti-clockwise or hold it's direction and move forward.
Rewards: A reward of 100-500 is given to the agent if the agent collides with/captures the evader. A small negative reward is also given for discouraging exploration of out of the boundaries.
The game episode ends when the evader is captured. The game is defined in such a way that the entire states/inputs given to the agent do not capture the entire environment states. For example, the pursuer may be in the field of the agent but when approached it might move away from the field. But since the evader moves away from the range of the pursuer, the states accessed by the pursuer do not reveal any information of the movement of the evader. Also there might be obstacles obstructing the view of the pursuer making the evader's position uncertain. So this game can be considered to be partially observable and represented by a POMDP. We used the following methods and techniques to handle these additional problems.

Methods and Techniques

We will briefly explain the main methods we used finally to deal with our application.

DQN with priority replay: In the usual DQN, we had to use the experience replay memory and target Q-networks to handle the stability issues which arise due to the neural network. But when we store our past experiences in the experience replay memory and sample it for training they are sampled randomly. But many a times especially for our case of pursuit evasion there would be long gaps before the evader gets detected or the evader gets captured which means then we would get plenty of experiences where the rewards are zero and the states would not change much. When they get stored we can see that the experiences which are important would be very scarce in the experience replay memory, because of which these experiences may go unsampled and eventually get evicted from the memory.

To combat this problem we prioritize the replay memory (Schaul et al. 2015). Thus, we would like our algorithm to sample the more important experiences more often. We take the experience with larger absolute TD-error(δ) to be more significant, as they symbolize that the networks have fit to those experiences most poorly.

$$\delta = |Y_t^{DDQN} - Q(S_t, A_t)|$$
$$Y_t^{DDQN} = R(S_t, A_t) + Q_{target}(S_{t+1}, argmax_a(Q(S_{t+1}, a))) \quad (4)$$

where Q_{target} refers to the Q-values output by the target network and Y_t^{DDQN} refers to the reward target generated by using the current DQN network to choose the next value from the target network. Notice that the update is different from that in the usual DQN algorithm, because of using the DoubleDQN (DDQN) (Van Hasselt, Guez, and Silver 2016) update. This helps in reducing overestimations during training. Each experience j in the memory has a corresponding absolute TD-error δ_j. We generate probabilities $P(j)$,

$$P(j) = \frac{\delta_j^\alpha}{\sum_k \delta_k^\alpha} \quad (5)$$

where α is a hyper parameter. With these probabilities, proper initialization of the new experiences and importance sampling weights to account for the bias introduced we are able to make sure that the more important experiences get sampled and get used for the Q-network loss minimization.

Deep Recurrent Q-Learning: As seen before the game forms a POMDP, and in order to handle this partial observability we use Deep Recurrent Q-Learning (Hausknecht and Stone 2015). As the evader keeps moving, from the definition of the states we can see that the agent does not get any sense of the direction, speed and the general movement of the evader. So to help the agent gather this temporal information we use Recurrent Neural Networks (RNNs) in our case, specifically the Long Short Term Memory (LSTM) networks.

The set of recent observations are input to the Q-network but the final linear layer is preceded by a LSTM of variable depth, to give us $Q(O, a; \theta)$, where O denotes the observations accessed by the agent and θ represents the network parameters. We try to learn the Q value as a function of the history of observations and actions. But in order to train this network we need to handle the hidden states of the LSTM. For training, we randomly sample a few episodes from the experience replay memory and choose a sequence of episode steps from each one of them and use these for training. But after each update operation we initialize the hidden state to zero, so that the hidden states are generated separately for each sequence of observations (Bootstrapped Random Updates). By this way of training using RNNs, we reduce the gap between $Q(O, a; \theta)$ and $Q(S, a; \theta)$, the Q-value over the complete environment states.

Experiments and Results

By applying the above methods and tuning for the hyperparameters we were able to train the single pursuer to learn to catch the evader for the more complicated game scenario when the evader does not just move randomly. In Figure 3. we have compared the performance of the deep recurrent Q-learning algorithm with LSTMs and a simple Q-learning algorithm with function approximation. We can see that the mean number of steps per episode decreases faster and reaches a much lower value than the linear function approximator case. So the agent is able to learn quickly and effectively with the help of the LSTMs.

Multi-Agent Pursuit Evasion

In the multi-agent version of pursuit evasion there are multiple pursuers trying to catch a single evader. We essentially want the pursuers to cooperate and collaborate to capture the evader. Construction of the simulator is similar to the single agent case but with the addition of new objects for the extra pursuers. Also the evader like in the single agent case does not just move randomly. Here it tries to move away from the

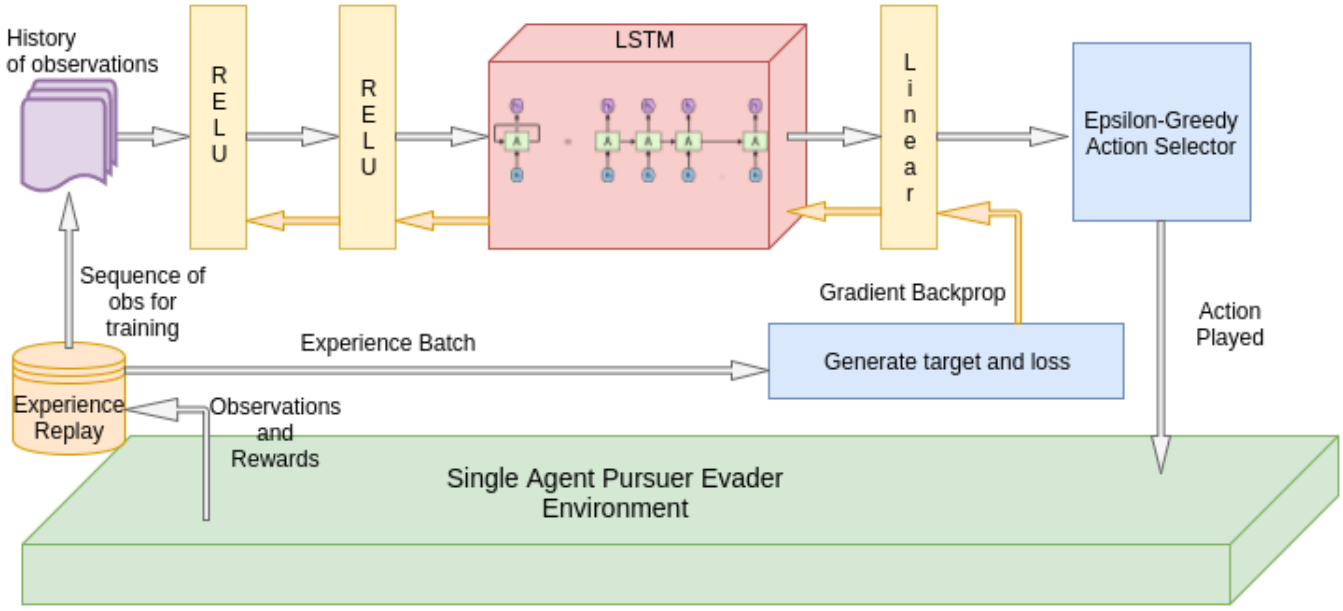

Figure 2: Architecture of the Deep Recurrent Q-Learning algorithm

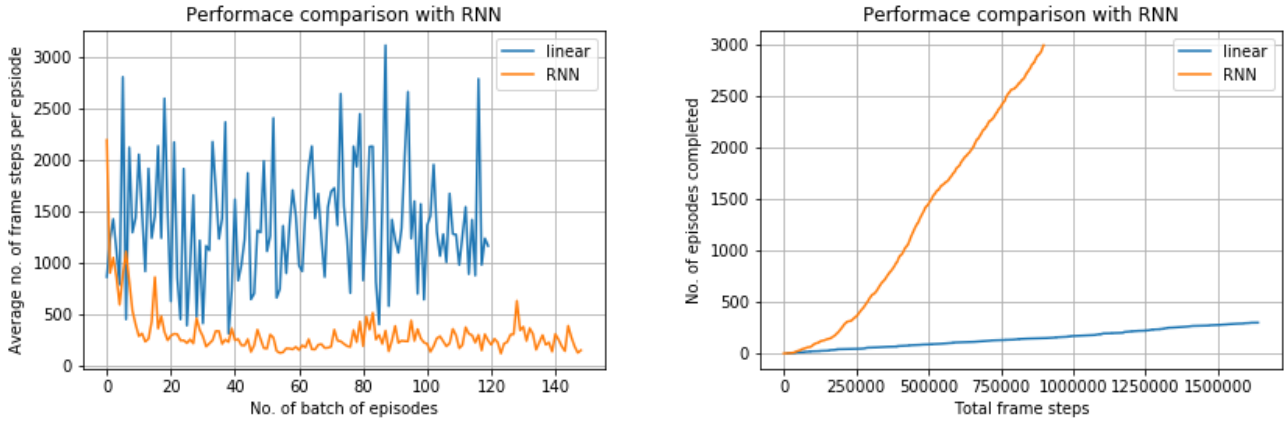

Figure 3: Performance using RNNs vs linear function approximator

closest pursuer to itself. For example, in the case of two pursuers and one evader, as the evader keeps moving away from the nearest evader, the only way to catch the evader would be for the pursuers to coordinate and attack the evader from opposite directions. The other game characteristics are similar to the single agent case. In the centralized case, the size of the state space and the action space increases exponentially with addition of new agents.

We now explain the methods we used to approach this problem.

Messaging with communication trade-off weights

In this setting we allow the agents to communicate with each other with small discrete messages using the RIAL algorithm (Foerster et al. 2016), and we want the agents to coordinate with each other to catch the evader. We try to make the agents learn to send messages and act according to the received messages. This is clearly a complicated scenario as not only must the agents learn to play good actions they must also learn to ignore wrong messages and interpret the right ones in order to choose better actions.

To make the agents learn this task we train the agents to send messages by using deep Q-learning. At any time instant each agent can transmit any one of the M discrete messages - $\{1, 2, ..., M\}$ to the other agents. So for each agent we create a separate Q-network that handles sending the messages to a corresponding teammate and there is another Q-network that handles the agent's actions.

Each agent i will have message networks representing $Q^m_{ij}(O_t, m'_t, m_t)$, the Q-value of sending message m_t from agent i to j, where O_t is the sequence of past observations received by the agent at step t and m'_t is the collection of all the messages sent to the agent by the other agents, which were generated in the previous time step. Likewise there is also the usual Q-network representing the Q-values for each action - $Q^a_i(O_t, m'_t, a_t)$. We can see that the Q-function depends not only on the environment observations but also the previous messages received. The input to the Q-networks is a concatenated version of O_t and m'_t. Since we use the DQN algorithm each network also has the corresponding target networks, representing $Q^a_{t_i}$ and $Q^m_{t_{ij}}$ respectively.

The agents choose the messages to send from the values Q^m_{ij} by using a message selector. In our case we use the same ϵ-greedy selector that we use for choosing the actions. Each agent plays an action, receives rewards from the environment and messages from the other agents, which are in turn used along with the past observation history to get the new actions and the messages to send. We store the experiences including the received messages in an experience replay memory. But it must be noted that the effects of the current messages sent will only be visible at the next time step. So to get the reward for the messages sent by an agent at a particular time step t, we must wait till the next time step to receive the next reward and this will be the corresponding message reward (R_{t+1}) for the previous time step. Using these separate message and action rewards, we store the experiences appropriately.

For applying the DQN updates we need to sample the experience replay by priority. But here for each agent i, as there are many Q-values and targets associated with the same experience we set the priority probabilities by using the total sum of absolute TD-errors (δ_i) considering the TD-errors of all the separate Q-values.

$$\delta_i = \delta^a_i + \sum_{j \neq i} \delta^m_{ij} \tag{6}$$

where

$$\begin{aligned} \delta^a_i &= |Y^a_i - Q^a_i(O_t, m'_t, a_t)|, \\ Y^a_i &= R_t + \gamma Q^a_{t_i}(O_{t+1}, m'_{t+1}, a''), \\ a'' &= argmax_{a^-}(Q(O_{t+1}, m'_t, a^-)), \end{aligned} \tag{7}$$

and

$$\begin{aligned} \delta^m_{ij} &= |Y^m_{ij} - Q^m_{ij}(O_t, m'_t, m_t)|, \\ Y^m_{ij} &= R_{t+1} + \gamma Q^m_{t_{ij}}(O_{t+1}, m'_{t+1}, m''), \\ m'' &= argmax_{m^-}(Q^m_{ij}(O_{t+1}, m'_{t+1}, m^-)), \end{aligned} \tag{8}$$

where δ^a_i correspond to the TD-error of the action network and δ^m_{ij} corresponds to the TD-error of the network handling agent i's messages to agent j. Now we use this total TD-error as in the single agent case for sampling the memory of each agent.

But with this method it can be seen that the TD-errors of both the message Q-network and the action Q-network are summed directly which can hinder learning of the task for the agents. Sometimes the TD-errors of the message network dominates and the algorithm picks these experiences with higher relative message network TD-errors. This makes the algorithm to focus too much on learning how to communicate at the cost of actually determining a good policy to act upon for the agents. Some other times the opposite also happens where the communication training gets ignored and the agents lack any coordination between them which again is undesirable. To combat this issue we introduce communication trade-off weights which balances this learning between communication between agents and their own action policies. We update the total TD-error as

$$\beta \delta^a_i + (1 - \beta) \sum_{j \neq i} \delta^m_{ij}, \tag{9}$$

where β is the communication trade-off weight, a hyper parameter, which can be adjusted in order to make either the action network or the message network more sensitive. If β is high, it stresses the experiences with larger action TD-errors to be considered more important, making the agent to concentrate more on training it's actions rather than the messages and vice versa if β is low.

By this method we train the multiple agents to learn to message each other and coordinate and choose actions to capture the evader. The psuedocode followed by each agent is given in Algorithm 1. The main architecture used in the algorithm can be seen in Figure 5.

Experiments and Results

We performed experiments mainly with two pursuers as more than two agents increased the training time drastically

Algorithm 1 Pseudocode for each messaging pursuer

//Assume parameter vectors θ_a and θ_m and global shared counter $T = 0$
//Assume target parameter vectors θ'_a and θ'_m
Load experience replay memory

repeat
  Set target parameters $\theta'_a \leftarrow \theta_a$ and $\theta'_m \leftarrow \theta_m$
  Load old experience tuple, old_exp
  repeat
    Get a_t and m_t with old_exp using ϵ-greedy selector
    Perform a_t and send m_t
    Receive reward r_t, new observations O_{t+1} and messages m'_{t+1}
    $current\_exp \leftarrow [O_t, r_t, a_t, O_{t+1}, m'_{t+1}, m_t]$
    Append r_t to old_exp as message reward
    Push old_exp onto replay buffer
    Pop oldest experience from replay buffer
    Sample mini-batch of experiences with $j \sim P(j) = \frac{\delta_j^\alpha}{\sum_k \delta_k^\alpha}$
    $T \leftarrow T + 1$
    Reset gradients: $d\theta_a \leftarrow 0$ and $d\theta_m \leftarrow 0$
    for each $exp \in$ minibatch **do**
      $[O_k, r_k^a, a_k, O_{k+1}, m'_{k+1}, m_k, r_k^m] \leftarrow exp$

$$R^a = \begin{cases} 0, & \text{for terminal } O_k \\ DDQN^a_{target}(O_{k+1}, m'_{k+1}), & \text{for non-termina} \end{cases}$$

$$R^m = \begin{cases} 0, & \text{for terminal } O_k \\ DDQN^m_{target}(O_{k+1}, m'_{k+1}), & \text{for non-termina} \end{cases}$$

      $DDQN^a_{gradient} \leftarrow$ Double-DQN gradient with target $r_k^a + \gamma R^a$
      $DDQN^m_{gradient} \leftarrow$ Double-DQN gradient with target $r_k^m + \gamma R^m$
      $d\theta_a \leftarrow d\theta_a + DDQN^a_{gradient}$
      $d\theta_m \leftarrow d\theta_m + DDQN^m_{gradient}$
      Update TD-targets δ for exp
      Update importance sample weights for exp
    end for
    Perform update on parameters θ_a using $d\theta_a$ and on θ_m using $d\theta_m$
    $old\_exp \leftarrow current\_exp$
  until episode terminates
until $T > T_{max}$

and also two agents were sufficient to learn to capture the evader. We used a network architecture for each pursuer similar to the single agent pursuer evader architecture and tuned the hyper parameters to make the messaging part work. In order to reduce the number of parameters, the message part and the action output part of the Q-networks both share the initial layers but have their own separate final linear layers. A β value of 0.7 was used finally after tuning.

In order to check the performance of the messaging method we tried to train the agents using the independent learners (Matignon, Laurent, and Le Fort-Piat 2012) method, where each agent would simply ignore the other agents and follow their own separate reinforcement learning algorithm. From Figure 6 we can see that with messaging using the trade-off weights the agents learn faster (the agent takes lesser number of frame steps to complete the episode) and we are able to get better results than with just the independent learners method. We also trained our agents with the centralized version of the Deep Q-Learning algorithm over the entire joint actions and joint policies. The results are shown in Figure 4. It can be seen that the messaging algorithm though understandably takes more steps to learn than the centralized case, it is still able to come close to the learning levels of the centralized case. Also the centralized version takes a lot more computational resources and time to train at each step due to the large number of parameters to train.

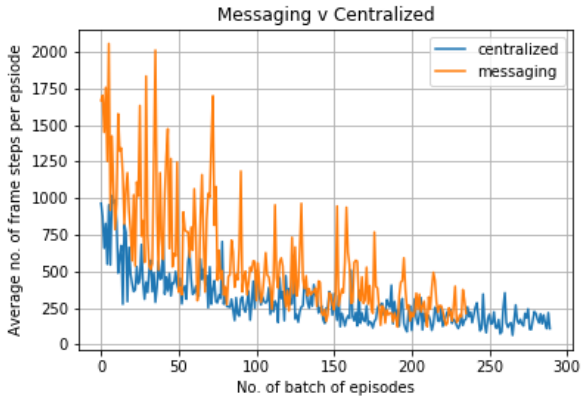

Figure 4: Comparing performance with centralized learning

Conclusion

In this paper, we solve the problem of pursuit evasion with techniques in Deep Reinforcement Learning. A simulator which allows us to construct complex pursuit evasion games with large state spaces was used for testing. In the single-agent case issues like partial observability were handled with LSTMs. For the multi-agent case we allowed the agents to message each other and introduced communication trade-off weights to balance the learning between co-ordination and self-action. With our proposed algorithm the agents learn to capture the evaders efficiently. The experimental results sug-

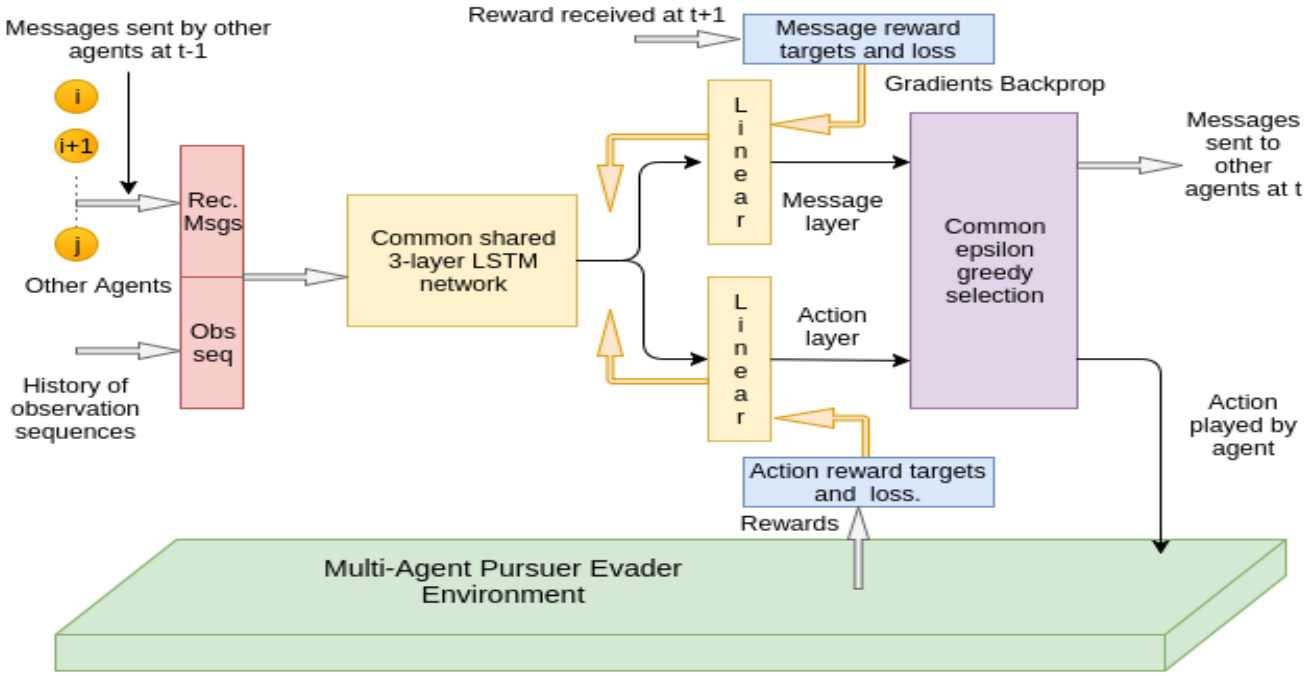

Figure 5: Architecture of the messaging algorithm

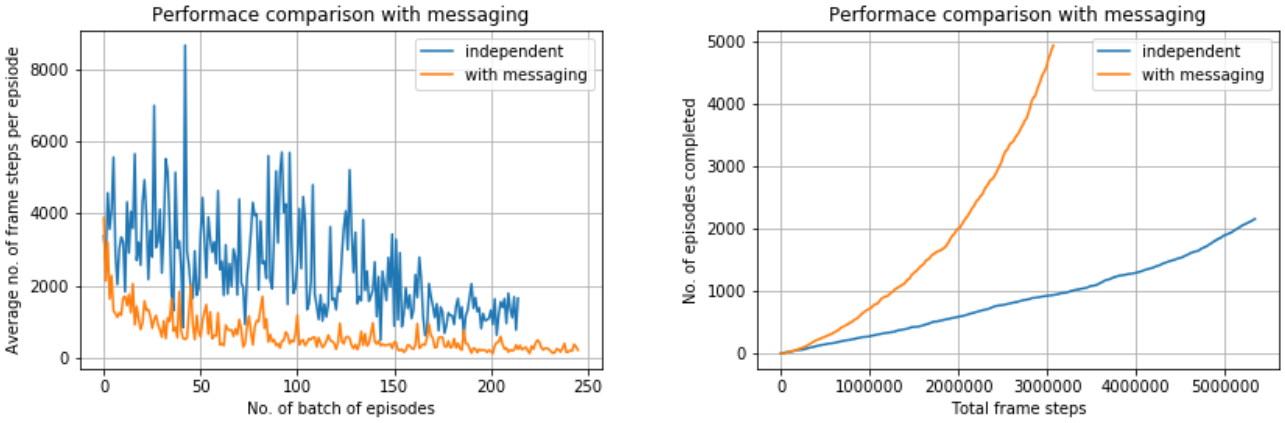

Figure 6: Performance comparison of messaging with DNNs and independent learners

gest that our algorithm for the multi-agent scenario can make agents learn their tasks better and faster without compromising too much on communication or action execution.

References

Bowling, M., and Veloso, M. 2001. Rational and convergent learning in stochastic games. In *International joint conference on artificial intelligence*, volume 17, 1021–1026. LAWRENCE ERLBAUM ASSOCIATES LTD.

Busoniu, L.; Babuska, R.; and De Schutter, B. 2008. A comprehensive survey of multiagent reinforcement learning. *IEEE Transactions on Systems Man and Cybernetics Part C Applications and Reviews* 38(2):156.

Foerster, J.; Assael, Y. M.; de Freitas, N.; and Whiteson, S. 2016. Learning to communicate with deep multi-agent reinforcement learning. In *Advances in Neural Information Processing Systems*, 2137–2145.

Hausknecht, M., and Stone, P. 2015. Deep recurrent q-learning for partially observable mdps. *arXiv preprint arXiv:1507.06527*.

Kapetanakis, S., and Kudenko, D. Reinforcement learning of coordination in cooperative multi-agent systems. *AAAI/IAAI* 2002:326–331.

Matignon, L.; Laurent, G. J.; and Le Fort-Piat, N. 2007. Hysteretic q-learning: an algorithm for decentralized reinforcement learning in cooperative multi-agent teams. In *Intelligent Robots and Systems, 2007. IROS 2007. IEEE/RSJ International Conference on*, 64–69. IEEE.

Matignon, L.; Laurent, G. J.; and Le Fort-Piat, N. 2012. Independent reinforcement learners in cooperative markov games: a survey regarding coordination problems. *The Knowledge Engineering Review* 27(01):1–31.

Mnih, V.; Kavukcuoglu, K.; Silver, D.; Rusu, A. A.; Veness, J.; Bellemare, M. G.; Graves, A.; Riedmiller, M.; Fidjeland, A. K.; Ostrovski, G.; et al. 2015. Human-level control through deep reinforcement learning. *Nature* 518(7540):529–533.

Schaul, T.; Quan, J.; Antonoglou, I.; and Silver, D. 2015. Prioritized experience replay. *arXiv preprint arXiv:1511.05952*.

Van Hasselt, H.; Guez, A.; and Silver, D. 2016. Deep reinforcement learning with double q-learning. In *AAAI*, 2094–2100.

Watkins, C. J., and Dayan, P. 1992. Q-learning. *Machine learning* 8(3-4):279–292.