Deep Reinforcement Learning Algorithms and their Applications

Rohit R.R
M.E S.S.A
SR No: 12321
Indian Institute of Science, Bangalore
Advisor : Prof.Shalabh Bhatnagar

***Abstract*—In this project we explore the various state of the art algorithms which use deep neural networks with Reinforcement Learning. Using deep neural networks in reinforcement learning algorithms enables us to successfully learn complex tasks directly from high dimensional sensory inputs. We try to improve upon the existing algorithms with experience replay, batch and weight normalization. The concept of weight normalization is shown to improve the performance of the algorithms. We then show an application of these algorithms on the problem of pursuit evasion with a single pursuer pursuing an evader, which we extend to the case of multi-agent pursuit evasion. We trained multiple agents to learn a complicated task of catching an evader by combining multi-agent reinforcement learning with deep reinforcement learning techniques.**

I. Introduction

Before the mid-term we had first analyzed the recent state of the art deep reinforcement learning algorithms like Deep Q-Learning [12], Asynchronous Actor Critic [11] and Trust Region Policy Optimization [15]. We discussed the advantages of handling large state spaces and complex tasks and also the constraints to overcome while using Deep Neural Networks (DNNs) with reinforcement learning algorithms. We tested the algorithms on many Atari Games and analyzed their relative performance. We explained our implementation and analysis and showed the results.

Next we moved on to improve upon the current algorithms. With the concept of weight normalization [13], which basically separates the weights into separate magnitude and direction variables so as to improve the conditioning of the gradient, we were able to improve upon the Asynchronous Actor Critic algorithm. We found that it improved the performance of the agents, mainly for the game of Breakout.

For the second part of our project which we have worked on after the midterm, we moved on to search for a good application suitable for these Deep Reinforcement Learning algorithms, make them work and analyze them. Since we have given a detailed view of the first part of our work in the mid-term report, we concentrate more on the second part in this report.

We briefly present the standard reinforcement learning setting to introduce the notations. 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 gives the agent a scalar reward r_t . The process continues until the agent reaches a terminal state after which the process restarts. The return $R_t = \sum_{k=0} \gamma^k r_{t+k}$ is the total accumulated return from time step t with discount factor $\gamma \in (0, 1)$. The goal of the agent is to maximize the expected return from each state s.
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 optimal value function $Q(s, a) = max_\pi Q^\pi(s, a)$ gives the maximum action value for state s and action a achievable by any policy. The value of state s under policy π is defined as $V^\pi(s) = E[R_t|s_t = s]$ and is simply the expected return for following policy π from state s. 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.

Since we used the Deep Q-Learning considerably more than the other algorithms for our post-midterm work, it is explained below briefly.
Deep Q-Learning: In this algorithm in [12], a deep neural network is used as a function approximator for the Q-Learning algorithm. But in order to tackle the issue of instability due to non IID data used for gradient updates, two methods - experience replay and target Q-network are

used.

Let $Q(s, a; \theta)$ be the approximate action value function with parameters θ represented by a DNN. We try to approximate the actual action value function, $Q(s, a) \approx Q(s, a; \theta)$. In one-step Deep Q-learning, the parameters θ of the action value function $Q(s, a; \theta)$ are learned by iteratively minimizing a sequence of loss functions, where the i^{th} loss function defined as

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

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, $\nabla_{\theta_i} L_i(\theta_i) =$

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

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

II. Pursuit Evasion Games

Pursuit evasion games are games where there are single/multiple pursuers which try to catch single/multiple evaders. Reinforcement learning is usually applied by taking the pursuer to be the agent and making it learn the appropriate policies to capture the evader. Because of its extensive applications, such as searching buildings for intruders, traffic control, military strategy,and surgical operation pursuit evasion seemed to be a worthy problem to deal with. Moreover, it was feasible to create a good simulator in Python making it a favourable application for the data hungry deep reinforcement learning algorithms. Since we create the simulator ourselves, it also provides the flexibility of defining the game in complicated ways to better test out the algorithms.

III. Construction of the simulator and the Game

The simulator for the pursuit evasion games was built with python. We used mainly the PYMUNK and PYGAME 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 detect the evader 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.

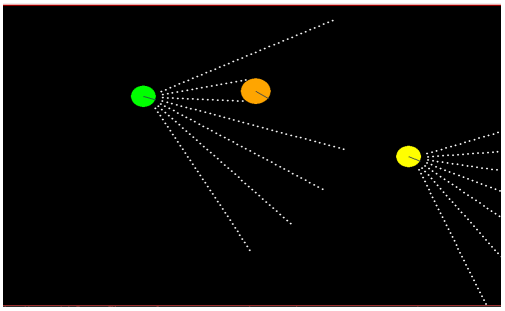

Fig. 1: Screen shot of the simulator

IV. 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 could see 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 mostly because the task to learn was simple even with the large state space. But when the game was modified such that the evader can evade the pursuer with a complicated 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 policy, the game gets considerably complicated 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.

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. So this game can be considered as a POMDP. As the game now is not only high dimensional and complex it is also not fully observable which means that we need something more than the deep Q-Learning algorithm used for Atari Games which has been shown in literature to work well only for fully observable scenarios. So we used the following methods and techniques to handle these additional problems.

A. Methods and Techniques

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

1) ***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 [14]. 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 important, as they symbolize that the networks have fit to those experiences most poorly.

$$\delta = |Y_t^{DDQN} - Q(S_t, A_t)|$$

$$Y_t^{DDQN} = R(S_t, A_t) + Q_{target}(S_{t+1}, argmax_a(Q(S_{t+1}, a)))$$

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) [16] update. This helps in reducing overestimations by decomposing the max operations. Each experience j in the memory has a corresponding absolute TD-error δ_j. We generate probabilities $P(j)$,

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

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.

2) ***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 [4]. 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.

B. Experiments and Results

By applying the above methods and tuning for the hyper-parameters 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.

V. Multi-Agent Reinforcement Learning Background

A stochastic game is represented by a tuple $< X, U_1, U_2, ..., U_n, f, \rho_1, \rho_2, ..., \rho_n >$ where n is the number of agents, X is the set of environment states, yielding

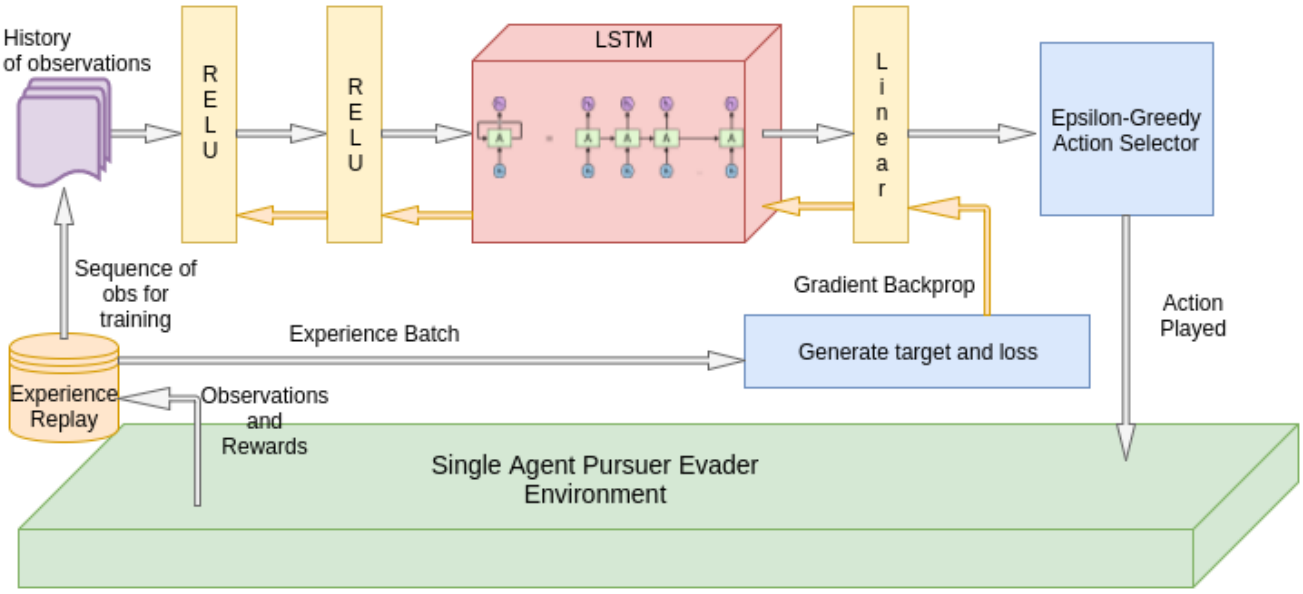

Fig. 2: Architecture of the Deep Recurrent Q-Learning algorithm

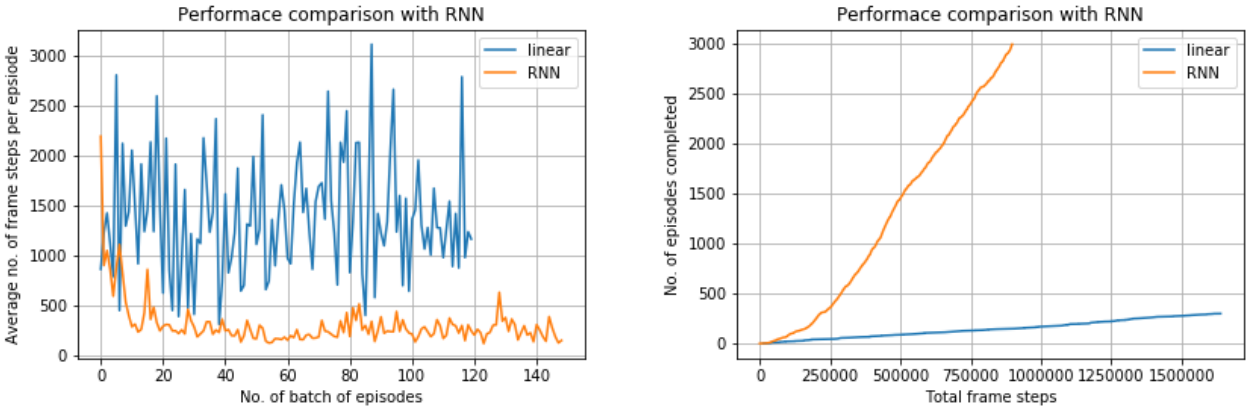

Fig. 3: Performance using RNNs vs linear function approximator

the joint action set $U = U_1 \times U_2 .... \times U_n$, $f : X \times U \times X \rightarrow [0, 1]$ is the state transition probability function, and $\rho_i : X \times U \times X \rightarrow \mathbb{R}, i = 1, 2, 3...., n$ 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 $h_i : X \times U_i \rightarrow [0, 1]$ form together the joint policy h. The Q-function, $Q_i^h : X \times U \rightarrow \mathbb{R}$ of each agent i for the multi-agent case depends on the policies of all the agents,

$$Q_i^h(x, u) = \mathbb{E}[\sum_k \gamma^k r_{i,k} | x_0 = x, h]$$

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/non-cooperative stochastic games.

In the case of centralized execution we solve MDPs/POMDPs over the joint actions and joint policies. In the decentralized execution 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 [2]. But we can use the methods in deep reinforcement learning to overcome these difficulties. We have concentrated more on the decentralized learning case since it is more challenging and is advantageous in terms of computation and speed.

VI. 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 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 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. For the decentralized case we considered both the cases where the agents did not communicate with each other and where they did with small discrete messages. If the agents can send anything between them, the situation becomes similar to centralized learning where only one agent can learn and send commands to the others. We wanted however the agents to learn a more difficult task.

A. Methods and Techniques

We looked into many multi-agent reinforcement learning algorithms like policy hill climbing [1], hysteric Q-learning [9], joint action learners [8], Nash Q-learning and more [2]. We tried many ways to combine multi-agent reinforcement learning and the recent deep reinforcement learning techniques to solve this multi-agent task. We now explain the two primary methods we used to approach this problem.

1) ***Joint Action Learners with DQN****:* Each agent has its own Q-network over the environment states and all actions, representing the individual Q-values, $Q_i(x, a_1, a_2, ..., a_n)$. The actions for each pursuer are the same as that of the single agent case while the states/observations for each pursuer in similar to the single agent pursuer states but concatenated with the discrete location values of each pursuer. Since we are dealing with the decentralized case each agent follows its own separate algorithms. So in order to apply the Deep Q-learning algorithm we need the Q-value over only the states and the agents' own action. This we get from the expected value (EV) [8] of each agent by summing over the probability of joint actions of the other agents.

$$EV(s, a^i) = \sum_{a^{-i} \in A_{-i}} Q(s, a^{-i} \cup a^i) \prod_{k \neq i} Pr^k(a^{-i}[k])$$

where $Pr^k(a^{-i}[k])$ is the probability of agent k choosing action $a^{-i}[k]$. In the literature usually each agent keeps a emperical distribution of the joint actions over each state. This is not suitable for us, due to the large state size of our application. So instead we used a softmax neural network to approximate the empirical distribution. We then train this network with the actions of the other agents as targets along with the usual DQN updates.

But this algorithm takes a long time to train. Usual Deep Q-Networks take the state as input and output the Q-values for each action. But in our case each agent has a Q-network outputting the Q-values for all possible joint action tuples. Addition of even a single agent requires training an entire new Q-network and the action space increases exponentially which in turn increases the network parameters to train for all the agents. Hence even though the algorithm showed progress while training it was becoming infeasible to train the agents well.

2) ***Messaging between agents****:* In this setting we allow the agents to communicate with each other with small discrete messages [3], 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, $Q^a_{t_i}$ 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.

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

where

$$\delta^a_i = |Y^a_i - Q^a_i(O_t, m'_t, a_t)|,$$

$$Y^a_i = R_t + Q^a_{t_i}(O_{t+1}, m'_{t+1}, a''),$$

$$a'' = argmax_{a^-}(Q(O_{t+1}, m'_t, a^-)),$$

and

$$\delta^m_{ij} = |Y^m_{ij} - Q^m_{ij}(O_t, m'_t, m_t)|,$$

$$Y^m_{ij} = R_{t+1} + Q^m_{t_{ij}}(O_{t+1}, m'_{t+1}, m''),$$

$$m'' = argmax_{m^-}(Q^m_{ij}(O_{t+1}, m'_{t+1}, m^-)),$$

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. As a variant to this, we update the total TD-error as

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

where β is 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.

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.

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 $j \sim P(j) = \frac{\delta^\alpha_j}{\sum_k \delta^\alpha_k}$
    $T \leftarrow T + 1$
    Reset gradients: $d\theta_a \leftarrow 0$ and $d\theta_m \leftarrow 0$
    for each $exp \in$ minibatch **do**
      $[O_k, r^a_k, a_k, O_{k+1}, m'_{k+1}, m_k, r^m_k] \leftarrow exp$

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

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

      $DDQN^a_{gradient} \leftarrow$
      Double-DQN gradient with target $r^a_k + \gamma R^a$
      $DDQN^m_{gradient} \leftarrow$
      Double-DQN gradient with target $r^m_k + \gamma R^m$
      $d\theta_a \leftarrow d\theta_a + DDQN^a_{gradient}$
      $d\theta_m \leftarrow d\theta_m + DDQN^m_{gradient}$
      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 $T > T_{max}$

B. Experiments and Results

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

In order to check the performance of the messaging method we tried to train the agents using the independent learners [10] 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 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.

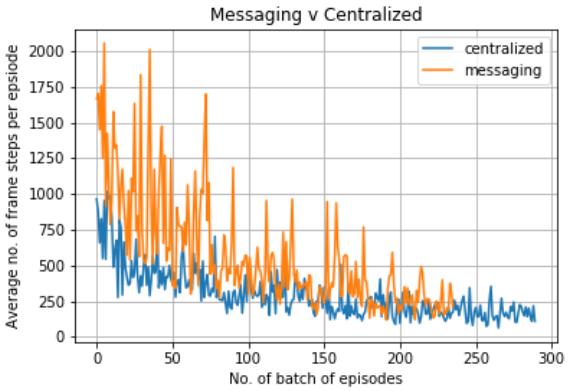

Fig. 4: Comparing performance with centralized learning

VII. Conclusion

The recent algorithms in Deep Reinforcement Learning to make agents learn to play Atari Games from raw screen images were explored. Improvements were made to the existing algorithms with weight normalization which enables conditioning of the gradient. There was significant performance improvement with this for the case of Breakout.

For the pursuer evader application, the use of LSTMs enabled us to overcome the partial observability problem in the game. The results show that the performance with these LSTMs and Deep reinforcement learning techniques enable us to overcome difficult hurdles which affect the usual Q-learning algorithms. Allowing multiple pursuers to message with discrete symbols using DNNs, we are able to make them coordinate and capture the evader. By training the messages and actions separately with shared networks and appropriate training methods, the agents learn to perform better than simply being independent learners. Overall the Deep Reinforcement Learning algorithms when applied suitably are powerful tools to handle high dimensional complex tasks in both single and multi-agent environments.

VIII. Future Work

In order to make pursuers to capture evaders in a real world scenario, sensors may not be of much use. We can mount cameras on the pursuers and process the video feed with conventional neural networks with LSTMs and then the discussed algorithms can be applied.

The evader can also be considered as capable of learning, in which case the evader would act as an adversary minimizing the pursuer's rewards. In such a case normal cooperative multi-agent reinforcement learning would not work. Hence other non-cooperative multi-agent reinforcement learning techniques like Nash Q-Learning can be used with the Deep Reinforcement Learning Techniques to handle the task.

References

[1] Michael Bowling and Manuela Veloso. "Rational and convergent learning in stochastic games". In: *International joint conference on artificial intelligence*. Vol. 17. 1. LAWRENCE ERLBAUM ASSOCIATES LTD. 2001, pp. 1021–1026.

[2] Lucian Busoniu, Robert Babuska, and Bart De Schutter. "A comprehensive survey of multiagent reinforcement learning". In: *IEEE Transactions on Systems Man and Cybernetics Part C Applications and Reviews* 38.2 (2008), p. 156.

[3] Jakob Foerster, Yannis M Assael, Nando de Freitas, and Shimon Whiteson. "Learning to communicate with deep multi-agent reinforcement learning". In: *Advances in Neural Information Processing Systems*. 2016, pp. 2137–2145.

[4] Matthew Hausknecht and Peter Stone. "Deep recurrent q-learning for partially observable mdps". In: *arXiv preprint arXiv:1507.06527* (2015).

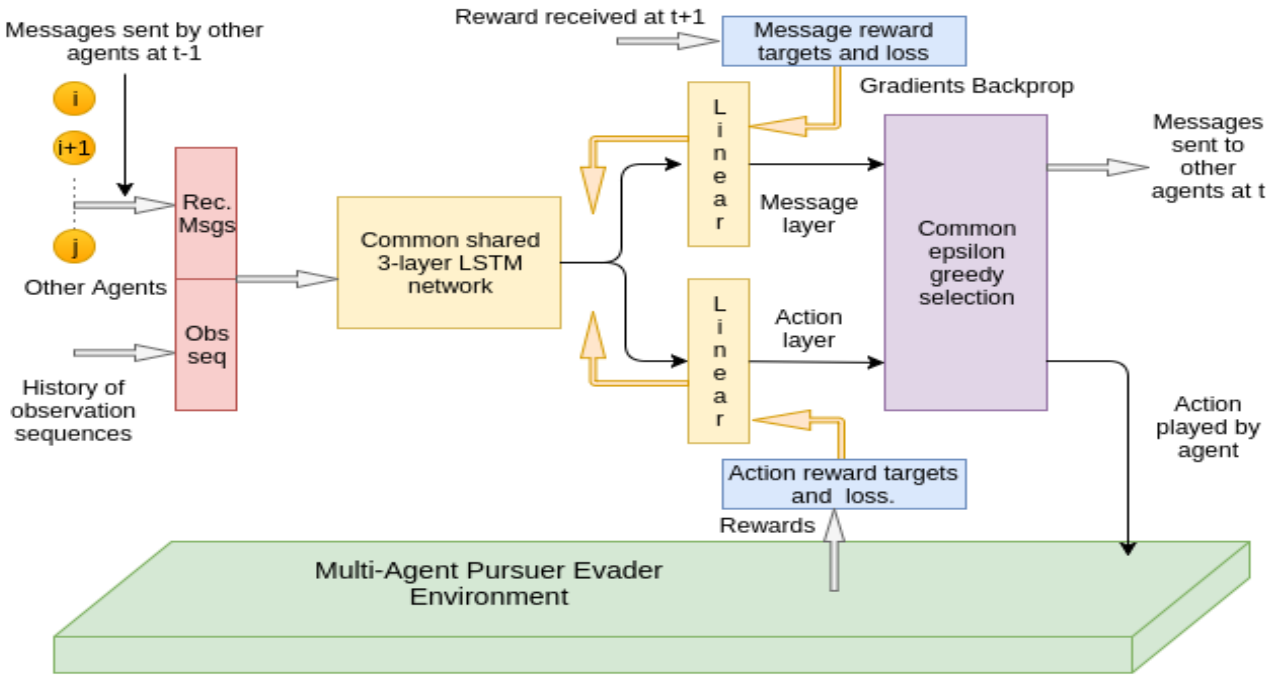

Fig. 5: Architecture of the messaging algorithm

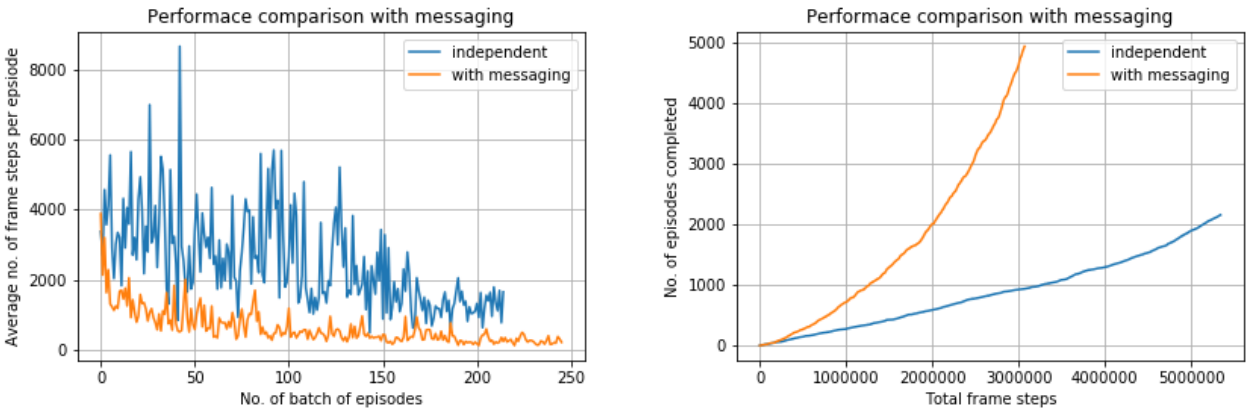

Fig. 6: Performance comparison of messaging with DNNs and independent learners

[5] Nicolas Heess, Jonathan J Hunt, Timothy P Lillicrap, and David Silver. "Memory-based control with recurrent neural networks". In: *arXiv preprint arXiv:1512.04455* (2015).

[6] Sergey Ioffe and Christian Szegedy. "Batch normalization: Accelerating deep network training by reducing internal covariate shift". In: *arXiv preprint arXiv:1502.03167* (2015).

[7] Sham Kakade and John Langford. "Approximately optimal approximate reinforcement learning". In: *ICML*. Vol. 2. 2002, pp. 267–274.

[8] Spiros Kapetanakis and Daniel Kudenko. "Reinforcement learning of coordination in cooperative multi-agent systems". In: *AAAI/IAAI* 2002 (), pp. 326–331.

[9] Laëtitia Matignon, Guillaume J Laurent, and Nadine Le Fort-Piat. "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*. IEEE. 2007, pp. 64–69.

[10] Laetitia Matignon, Guillaume J Laurent, and Nadine Le Fort-Piat. “Independent reinforcement learners in cooperative Markov games: a survey regarding coordination problems”. In: *The Knowledge Engineering Review* 27.01 (2012), pp. 1–31.

[11] Volodymyr Mnih, Adria Puigdomenech Badia, Mehdi Mirza, Alex Graves, Timothy P Lillicrap, Tim Harley, David Silver, and Koray Kavukcuoglu. “Asynchronous methods for deep reinforcement learning”. In: *arXiv preprint arXiv:1602.01783* (2016).

[12] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A Rusu, Joel Veness, Marc G Bellemare, Alex Graves, Martin Riedmiller, Andreas K Fidjeland, Georg Ostrovski, et al. “Human-level control through deep reinforcement learning”. In: *Nature* 518.7540 (2015), pp. 529–533.

[13] Tim Salimans and Diederik P. Kingma. “Weight Normalization: A Simple Reparameterization to Accelerate Training of Deep Neural Networks”. In: *CoRR* abs/1602.07868 (2016).

[14] Tom Schaul, John Quan, Ioannis Antonoglou, and David Silver. “Prioritized experience replay”. In: *arXiv preprint arXiv:1511.05952* (2015).

[15] John Schulman, Sergey Levine, Philipp Moritz, Michael I. Jordan, and Pieter Abbeel. “Trust Region Policy Optimization”. In: *CoRR* abs/1502.05477 (2015).

[16] Hado Van Hasselt, Arthur Guez, and David Silver. “Deep Reinforcement Learning with Double Q-Learning.” In: *AAAI*. 2016, pp. 2094–2100.

[17] Ronald J Williams. “Simple statistical gradient-following algorithms for connectionist reinforcement learning”. In: *Machine learning* 8.3-4 (1992), pp. 229–256.