Deep Reinforcement Learning Algorithms and their Applications

A PROJECT REPORT

SUBMITTED IN PARTIAL FULFIMENT OF THE

REQUIREMENTS FOR THE DEGREE OF

Master of Engineering

IN

System Science and Automation

BY

Rohit R.R

Department of Electrical Engineering
Indian Institute of Science
Bangalore – 560 012 (INDIA)

June, 2017

DEDICATED TO

My Parents

and

Friends

Acknowledgements

I take this opportunity to express my gratitude to the people who have helped and supported me throughout my Masters. Foremost, I would like to thank my project advisor Prof.Shalabh Bhatnagar, for giving me the freedom to do my project in a relatively new field and encouraging me to follow up on my ideas even when I was facing troubles with parts of my project. Without his guidance and valuable inputs the project would not have proceeded as it did.

I also would like to thank the Professors of EE and CSA departments whose courses helped me to learn the topics required to deal with this project. I am thankful to the members of the Stochastic Systems lab who helped me by sharing their knowledge and kept me motivated during the course of my project. I am also thankful to the non-technical staff of EE and CSA departments and the hostel and mess workers who have made my stay in IISc a pleasant one.

I would like to thank my friends for providing a helping hand in most of my endeavours and for being a relaxing influence at times of stress. Most importantly, I thank my parents for their constant love and support, without which none of this would have been possible.

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. The state of the art deep reinforcement learning algorithms are analyzed with Atari Games. 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. These algorithms are applied on the problem of pursuit evasion. First, we solve the problem of single agent pursuit evasion using Recurrent Neural Networks and Priority Replay. Then for the problem of multi-agent pursuit evasion, we train multiple agents to learn a complicated coordinated task of catching an evader by allowing them to communicate with each other through discrete messages and using multi-agent reinforcement learning algorithms along with deep reinforcement learning techniques.

Contents

List of Figures

Notations and Abbreviations

RL - Reinforcement Learning
DNN - Deep Neural Networks
LSTM - Long Short Term Memory
DQN - Deep Q-Network
MDP - Markov Decision Process
POMDP - Partially Observable Markov Decision Process
RMS - Root Mean Square
IID - Identically and Independently Distributed

Chapter 1

Introduction

Reinforcement Learning is a field of machine learning, concerned with how agents ought to take actions in an environment so as to maximize some notion of cumulative reward. But in order to use reinforcement learning in most real-life cases with high complexity we mostly need to deal with high dimensional state spaces and we must derive efficient representations from the high dimensional sensory inputs that we get. So while reinforcement learning has been successful in many cases, their applicability has been restricted to cases where useful features can be handcrafted or in cases with fully observed, low dimensional state spaces. But with the advent of Deep Neural Networks and with their power of learning these constraints are being overcome. Combining reinforcement learning algorithms with Deep Learning concepts give rise to "Deep Reinforcement Algorithms".

We wanted to analyze these new algorithms, to check their performance when compared to the previous reinforcement learning algorithms. The Atari Games provided us a good test bed to analyze these algorithms due to the availability of software libraries that helped in the implementation. There are few key issues in using Deep Neural Nets (DNNs) with reinforcement learning algorithms. One is that the data/experiences generated in a reinforcement learning scenario are correlated. Using this data to perform stochastic gradient updates makes it difficult for the neural networks to converge making the learning unstable. So when fitting/optimizing a neural network by stochastic gradient methods we preferrably want to use IID data. Secondly, reinforcement Learning is generally not a very

fast learning method involving iterative updates of an evaluation followed by some improvement. Couple this with the high training time and data required for deep neural networks, we can see that scaling these algorithms becomes an issue. They would take a long time to train and would require huge amounts of data. So in order to make some improvements to the existing algorithms concepts like experience replay for stabilizing the networks, batch and weight normalization which have shown great results in improving training of the Deep Neural Networks are used.

After getting a proper hold of these algorithms we looked to apply it on some suitable applications . So we chose this problem of Pursuit Evasion. Pursuit Evasion games have extensive applications in searching buildings for intruders, traffic control, military strategy,and surgical operation. It was also feasible to make a simulator for the problem making it an ideal candidate to apply the deep reinforcement learning algorithms. These games include the single-agent case and the multi-agent case. In the single agent case issues like partial observability occur due to the absence of evader information at all times. In the multi-agent case the agents need to cooperate with each other to catch the evaders. Accomplishing this in a high dimensional state space setting has been difficult. So in order to combat such problems we try to use the memory property of Recurrent Neural Networks (RNNs) and allow messaging between agents.

We start with explaining some of the reinforcement learning and multi-agent reinforcement learning concepts used for the algorithms. Then we explain the deep reinforcement algorithms that have been introduced recently mainly for discrete action spaces, which are followed by the improvements we tried to make on the algorithms. Then we proceed to introducing the pursuer evader game scenario and explain the methods we used to solve the single and multi-agent pursuer evader games.

Chapter 2

Background

2.1 Reinforcement Learning

We consider the standard reinforcement learning setting where an agent interacts with an environment E over number of discrete time steps. At each time step t, the agent receives a state s_t and selects an action at from some 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. Similarly, 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.

There are model based and model free methods in reinforcement learning but since most popular effective algorithms are based on model free techniques we stick to that.

In value-based model-free reinforcement learning, the methods involve solving

the Bellman equation iteratively to arrive at the solution. In these methods the action value function is represented using a function approximator, such as a neural network. Let $Q(s, a; \theta)$ be an approximate action-value function with parameters θ. The updates to θ can be derived from a variety of reinforcement learning algorithms. One example of such an algorithm is Q-learning, which aims to directly approximate the optimal action value function:$Q(s, a) \approx Q(s, a; \theta)$. In one-step 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

$$L_i(\theta_i) = \mathbb{E}(r + \gamma \max_{a'} Q(s', a'; \theta_{i-1}) - Q(s, a; \theta_i))^2$$

where s' is the state encountered after state s. We refer to this method as one-step Q-learning because it updates the action value $Q(s, a)$ toward the one-step return $r + \gamma \max_a Q(s, a; \theta)$.

In contrast to value-based methods, policy-based model-free methods directly parameterize the policy $\pi(a|s; \theta)$ and update the parameters π by performing, typically approximate, gradient ascent on $\mathbb{E}[R_t]$. One example of such a method is the REINFORCE family of algorithms due to Williams (1992) [18]. Standard REINFORCE updates the policy parameters θ in the direction $\nabla_\theta log\pi(a_t|s_t; \theta)R_t$, which is an unbiased estimate of $\nabla_\theta \mathbb{E}[R_t]$. It is possible to reduce the variance of this estimate while keeping it unbiased by subtracting a learned function of the state $b_t(s_t)$, known as a baseline (Williams, 1992), from the return. The resulting gradient is $\nabla_\theta log\pi(a_t|s_t; \theta)(R_t - b_t(s_t))$. A learned estimate of the value function is commonly used as the baseline $b_t(s_t) \approx V_\pi(s_t)$ leading to a much lower variance estimate of the policy gradient. When an approximate value function is used as the baseline, the quantity $R_t - b_t$ used to scale the policy gradient can be seen as an estimate of the advantage of action a_t in state s_t, or $A(s_t, a_t) = Q(s_t, a_t)V(s_t)$, because R_t is an estimate of $Q_\pi(s_t, a_t)$ and b_t is an estimate of $V_\pi(s_t)$. This approach can be viewed as an actor-critic architecture where the policy π is the actor and the baseline b_t is the critic.

2.2 Multi-Agent Reinforcement Learning

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 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 most cases the optimal Q-function for each agent is learnt to get the optimal policies for all agents. 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.

Chapter 3

Benchmark for Deep RL

Deep Reinforcement Learning algorithms are usually tested out on game environments like Atari games and Physics MuJoCo Simulators (simulations on learning to walk) due to two main reasons. One is that these learning tasks have proved to be difficult for the traditional reinforcement learning algorithms due to the high dimensionality of the state-action spaces and due to the complexity of the learning tasks, so they would prove to be challenging for any learning algorithm. Secondly there are many readily available game engines written in different languages that provide simple clean interfaces which makes it flexible and easy to test out all aspects of an algorithm. Figure 3.1 shows the screenshots of some Atari games.

Figure 3.1: Screenshots of popular games like Breakout and Space Invader-on the first row

Chapter 4

Related Work

In this recent domain of deep reinforcement learning there have been three main approaches/algorithms introduced which have great success for discrete action space games especially. Continuous action spaces are more difficult to deal with often requiring a lot of computational resources [9][5]. Also another algorithm considering the Atari game as a Partially Observable MDP has also shown good results. A short overview of the recent state of the art Deep RL algorithms is given below.

4.1 Deep Q-Learning

In this algorithm in [12], a convolutional 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 are used.

Experience Replay: The experiences (including the state, action taken in the state and reward obtained) are saved in a replay memory buffer. The buffer is filled and if any new experience arrives the old ones are popped out. Now instead of using the immediate experiences for the gradient updates as in the case of the normal Q-Learning algorithm here a batch of experiences are sampled from the replay memory and then this data is used for the gradient updates. This helps in ensuring that the gradient updates are sufficiently uncorrelated as experiences sampled are mostly far apart and not related to each other.

Target network: In the Q-Learning algorithm the Q function approximator used to choose an action, also generates the TD targets and the updates that we

get as a result of this are used to update the same function approximator. This again induces correlation in the updates. When $Q(s_t, a_t)$ is increased, $Q(s_{t+1}, a)$ for all a also would mostly change in the same way hence increasing the TD targets which might lead to divergence of the networks. To combat this two separate networks, one target network - the network that would generate the TD targets for the update and the normal network - network which chooses action, where the gradient updates happen are used for every step. After every few steps the parameters of the updated network are copied on to the target network. This ensures that the updates do not affect the targets immediately ensuring more stability in learning.

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

On differentiation $\nabla_{\theta_i} L_i(\theta_i) =$

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

The above equations show the loss function and the corresponding gradients w.r.t parameters. θ^- represents the parameters of the target network. The batch of gradients are then used for the Q-learning updates.

4.2 Asynchronous Actor-Critic

In this algorithm in [13] multiple agents try to learn simultaneously using Actor Critic updates. Each agent corresponds to a thread and we spawn multiple threads that perform operations asynchronously and parallely. Each thread has their own network (value network and policy network). There's a common shared parameter server which stores a copy of network parameters. Now as the agents run the actor critic algorithm they generate gradients which would be used to change the network parameters using some stochastic gradient updates. But instead of updating their own parameters the threads send their updates to a common parameter server where the parameters are updated.

All these operations are performed asynchronously and parallely meaning the threads are not synchronized while updating the common parameter server. So some updates may collide and we may lose those updates but because of the

HogWild property mentioned in [13] in an expected sense we would still be able to learn well. Before performing a set of actions again to get the next updates each thread synchronizes its parameters with the shared parameters .

The parallelism in the algorithm serves two purposes. One is that it increases the training speed as more training steps are done with more agents. Secondly the parallel agents will more likely be seeing different parts of the state space at any point of time as they follow stochastic policies, so the updates they make would be independent of each other thus overcoming the issue of divergence due to non IID updates. Following are the actor-critic gradient accumulation updates performed by each agent.

$$d\theta \leftarrow d\theta + \nabla_{\theta'} log\pi(a_i|s_i;\theta')(R - V(s_i;\theta'_v))$$

$$d\theta_v \leftarrow d\theta_v + \partial(R - V(s_i;\theta'_v))^2/\partial\theta'_v$$

For Atari games in [13] and [12], four consecutive screen frames of the game are taken to be a state. This ensures that the state is able to capture the temporal aspects of the game, for example if a ball is moving in a game like breakout we can get a sense of direction and speed with which the ball is moving by using the four frames. So the Atari games are formulated as fully observable MDPs and use the algorithm. A convolutional neural network is used as a function approximator that enables them to learn good representations of the environment.

4.3 Trust Region Policy Optimization

In the usual policy iteration and policy gradient algorithms a lot of learning and unlearning happens meaning every update does not necessarily maximize the reward function. So in [16] they have tried to overcome this by attempting to make every update to actually improve on the rewards we get. Using the identity,

$$\eta(\widetilde{\pi}) = \eta(\pi) + \mathbb{E}_{r\sim\widetilde{\pi}}[A^{\pi}(s_0, a_0) + A^{\pi}(s_1, a_1) + ...]$$

where $\eta(\pi) = \mathbb{E}[R|\pi]$ and $A(s, a)$ is the advantage function. Considering a parameterized policy, making some approximations we get a first order approximation

$L(\theta)$ to $\eta(\pi)$ in parameter θ (full theoretical proof in [16]).

$$L_{\theta_{old}} = \mathbb{E}_s\Big[\sum_{t=0}^{T-1} \mathbb{E}_{a\sim\theta}[\frac{\pi(a_t|s_t,\theta)}{\pi(a_t|s_t,\theta_{old})} A^{\theta}(s_t,a_t)]\Big]$$

Then using conservative policy iteration in [7] a theorem in [16] shows the following,

$$\eta(\theta) \geq L_{\theta_{old}} - C \max_s D_{KL}[\pi(\cdot|\theta_{old},s)||\pi(\cdot|\theta,s)]$$

where D_{KL} is the Kullback-Leibler divergence between probability distributions. So the basic concept used is to optimize the right hand side which will increase the total reward. But in order to make this algorithm practical many more approximations are made [16].

Chapter 5

Proposed Modifications to Deep RL algorithms

Many ideas and approaches were tried to improve upon these Deep reinforcement learning algorithms. As while analyzing these algorithms we found the Asynchronous Actor Critic algorithm to perform better than the other algorithms for the case of Atari games, we stuck to improving upon this algorithm mostly. So the following modifications were mainly applied to the Asynchronous Actor Critic algorithm.

5.1 Experience Replay

One of the main concerns with deep reinforcement learning is that the data used to perform gradient updates may get correlated which destabilizes the network. So to avoid this, experience replay was used in [12] and parallelism was necessary in [13]. To improve the stability we combined this experience replay concept in the Asynchronous Actor Critic Algorithm. So a replay memory buffer was given to each agent/thread from which experiences were stored and sampled for the updates. Algorithm 1 with the same architecture as before was used by each actor learner process.

Observation : Using this concept did not improve the training and results much. This might be because the parallelism itself made sure that the data is IID enough. Also the experience replay memory used was small due to memory constraints.

Plus addition of these operations slows the algorithm further making it not worth the stabilization it provides if any. But we feel that this concept of experience replay would be very helpful on some games (maybe some more complicated games) where the agents are not able to explore the state space well, in which case the data would be correlated and hence this would be useful.

5.2 Batch Normalization

Batch Normalization introduced in [6] has shown tremendous emperical results in improving the training speeds of Deep Neural Networks. An internal covariance shift (the change in the distribution of network activations due to the change in network parameters during training) occurs for each subnetwork, which makes training difficult as parameters need to change to considering these shifts. To handle this the input to each activation is normalized as follows:
For a layer with d-dimensional input $x = (x^{(1)}, ..., x^{(d)})$, we normalize each dimension,

$$\widetilde{x}^{(k)} = \frac{x^{(k)} - E[x^{(k)}]}{\sqrt{Var[x^{(k)}]}}$$

For instance, normalizing the inputs of a sigmoid would constrain them to the linear regime of the nonlinearity. To address this, it is made sure that the transformation inserted in the network can represent the identity transform. To accomplish this, for each activation x^k , a pair of parameters are introduced γ^k, β^k, which scale and shift the normalized value: $y^k = \gamma^k x^k + \beta^k$. These parameters are learned along with the original model parameters, and restore the representation power of the network. This is applied for each mini-batch during training to make it practical.

Issues : But on applying this to networks in Deep RL the algorithms seem to actually worsen the training rather than improve it. The networks most often seem to diverge after some time else do not seem to learn well at all. This might be because in [6] it has been tried mainly on supervised learning scenarios where the training inputs are usually IID. But with reinforcement learning this is usually not the case. So the dependent correlated inputs probably are causing this instability.

Algorithm 1 Pseudocode for each actor-learner process with experience replay

//Assume global shared parameter vectors θ and θ_v and global shared counter $T = 0$

// Assume thread-specific parameter vectors θ' and θ'_v

Initialize thread step counter $t \leftarrow 1$ and load experience replay memory

repeat

Reset gradients : $d\theta \leftarrow 0$ and $d\theta_v \leftarrow 0$

Synchronize thread-specific parameters $\theta' = \theta$ and $\theta'_v = \theta_v$

$t_{start} = t$

Get state s_t

repeat

Perform a_t according to policy $\pi(a_t|s_t;\theta')$

Receive reward r_t and new state s_{t+1}

Pop oldest experience from replay buffer

Push $[s_t, a_t, r_t, V(s_t;\theta'_v)]$ in replay buffer

$t \leftarrow t + 1$

$T \leftarrow T + 1$

until terminal s_t or $t - t_{start} == t_{max}$

$$R = \begin{cases} 0, & \text{for terminal } s_t, \\ V(s_t, \theta'_v), & \text{for non-terminal state } s_t \end{cases}$$

Choose a random integer t_e from $[0, \text{replay memory size-1} - t_{max}]$

for $i \in \{t_e, ..., t_e + t_{max}\}$ **do**

Take experience i from buffer -

$[r_i, s_i, a_i, V(s_i;\theta'_v)]$

$R \leftarrow r_i + \gamma R$

Accumulate gradients wrt θ' :

$d\theta \leftarrow d\theta + \nabla_{\theta'} log\pi(a_i|s_i;\theta')(R - V(s_i;\theta'_v))$

Accumulate gradients wrt θ'_v :

$d\theta_v \leftarrow d\theta_v + \partial(R - V(s_i;\theta'_v))^2/\partial\theta'_v$

end for

Perform asynchronous update on shared parameters θ using $d\theta$ and of θ_v using $d\theta_v$

until $T > T_{max}$

5.3 Weight Normalization

The weight normalization concept in [14] was seen to improve the training speed in deep neural networks that deals with noisy data. So it could be used on the deep RL networks when considering the non iid behaviour of the data in the RL scenario to be noisy.

For this weight normalization the weight vectors that deal with inputs to each activation function/node are taken to be the product of a magnitude and a unit vector which are controlled separately.

$$w = \frac{g}{||v||}v$$

where g is a scalar and v is a vector and so
$||w|| = g$.

Here [14] proposes to explicitly reparameterize the model and to perform stochastic gradient descent in the new parameters v, g directly. Doing so improves the conditioning of the gradient and leads to improved convergence of the optimization procedure. By decoupling the norm of the weight vector (g) from the direction of the weight vector ($v/||v||$), the convergence of the stochastic gradient descent optimization speeds up.

Reparameterizing the network increases the number of variables. But despite this we can get direct equations relating the gradients of weight vector w with those of v and g as shown below. Hence it does not require any extra computation for getting the gradients of the added variables.

$$\nabla_g L = \frac{\nabla_w L.v}{||v||}, \quad \nabla_v L = \frac{g}{||v||}\nabla_w L - \frac{g\nabla_g L}{||v||^2}v$$

The psuedocode for this method can be seen in Algorithm 2 where θ_v and θ are merged to get w. The same network architecture used for asynchronous actor critic implementation is used by each process.

Algorithm 2 Pseudocode for each actor-learner process with weight normalization

//Assume global shared parameter vectors θ and θ_v and global shared counter $T = 0$

// Assume thread-specific parameter vectors θ' and θ'_v

Initialize thread step counter $t \leftarrow 1$ and load experience replay memory

repeat

Reset gradients : $d\theta \leftarrow 0$ and $d\theta_v \leftarrow 0$

Synchronize thread-specific parameters $\theta' = \theta$ and $\theta'_v = \theta_v$

Convert to network weights with $w = \frac{g}{||v||} v$ $t_{start} = t$

Get state s_t

repeat

Perform a_t according to policy $\pi(a_t|s_t;\theta')$

Receive reward r_t and new state s_{t+1}

$t \leftarrow t + 1$

$T \leftarrow T + 1$

until terminal s_t or $t - t_{start} == t_{max}$

$$R = \begin{cases} 0, & \text{for terminal } s_t, \\ V(s_t, \theta'_v), & \text{for non-terminal state } s_t \end{cases}$$

for $i \in \{t-1, ..., t_{start}\}$ **do**

$R \leftarrow r_i + \gamma R$

Accumulate gradients wrt θ' :

$d\theta \leftarrow d\theta + \nabla_{\theta'} log\pi(a_i|s_i;\theta')(R - V(s_i;\theta'_v))$

Accumulate gradients wrt θ'_v :

$d\theta_v \leftarrow d\theta_v + \partial(R - V(s_i;\theta'_v))^2/\partial\theta'_v$

end for

Convert to gradients wrt g, v $\nabla_g L = \frac{\nabla_w L.v}{||v||}, \quad \nabla_v L = \frac{g}{||v||}\nabla_w L - \frac{g\nabla_g L}{||v||^2} v$

Perform asynchronous update on shared parameters using $\nabla_g L$ and $\nabla_v L$

until $T > T_{max}$

Chapter 6

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.

6.1 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 6.1 represent the pursuers and the or-

ange 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. The pursuers are controlled by our policies, while the evader can be defined to follow any behaviour that we would like it to have. For the most part of our project the evader tries to move directly away from the pursuer when the pursuer comes too close.

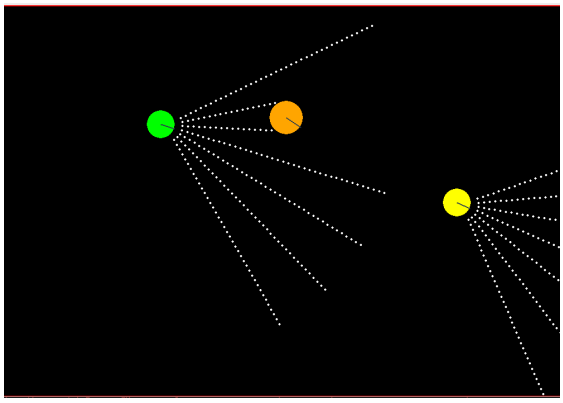


Figure 6.1: Screen shot of the simulator

Chapter 7

Single-Agent Pursuit Evasion

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 and it's own location. 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 on the speed and the direction movement of the evader. As the states/observations received by the agent does not capture the entire game characteristics, this setting 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. While trying to solve using the Asynchronous Actor Critic algorithm it was difficult to make the networks converge. For this reason we mostly stuck to using Deep Q-Learning and it's variants to solve both the single and multi-agent cases. We used the following methods and techniques to handle these additional problems.

7.1 Methods and Techniques

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

7.1.1 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 [15]. 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) [17] 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.

7.1.2 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. The input here would be a sequence of past observations instead of just one. We try to learn the Q value as a function of the history of observations

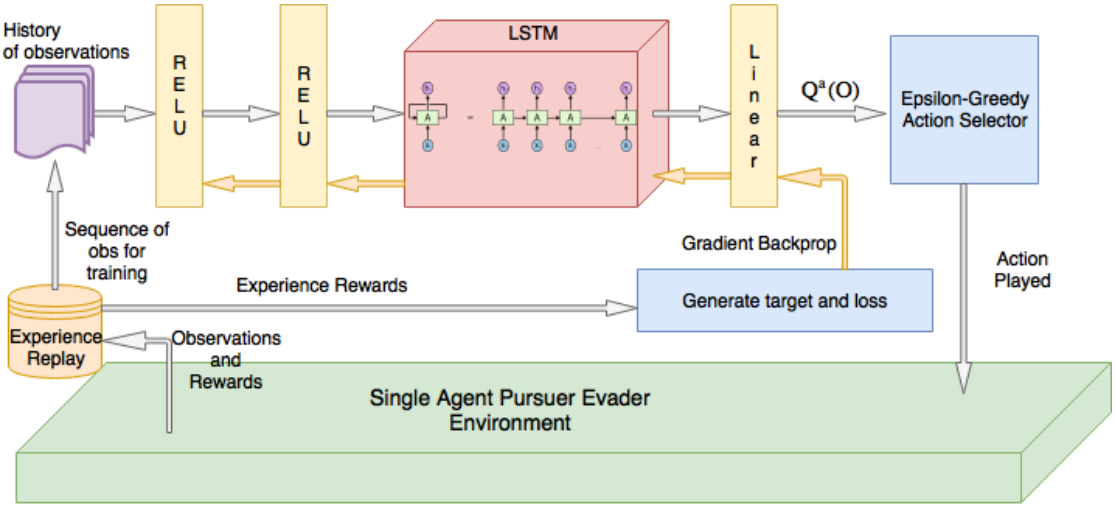


Figure 7.1: Architecture of the Deep Recurrent Q-Learning algorithm

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. The trained LSTMs then help to consider the temporal aspects of the game to give better policies. The architecture used for training the single agent pursuit evasion case can be seen in Figure 7.1.

Chapter 8

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.

8.1 Methods and Techniques

We looked into many multi-agent reinforcement learning algorithms like policy hill climbing [1], hysteric Q-learning [10], 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.

8.1.1 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.

8.1.2 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

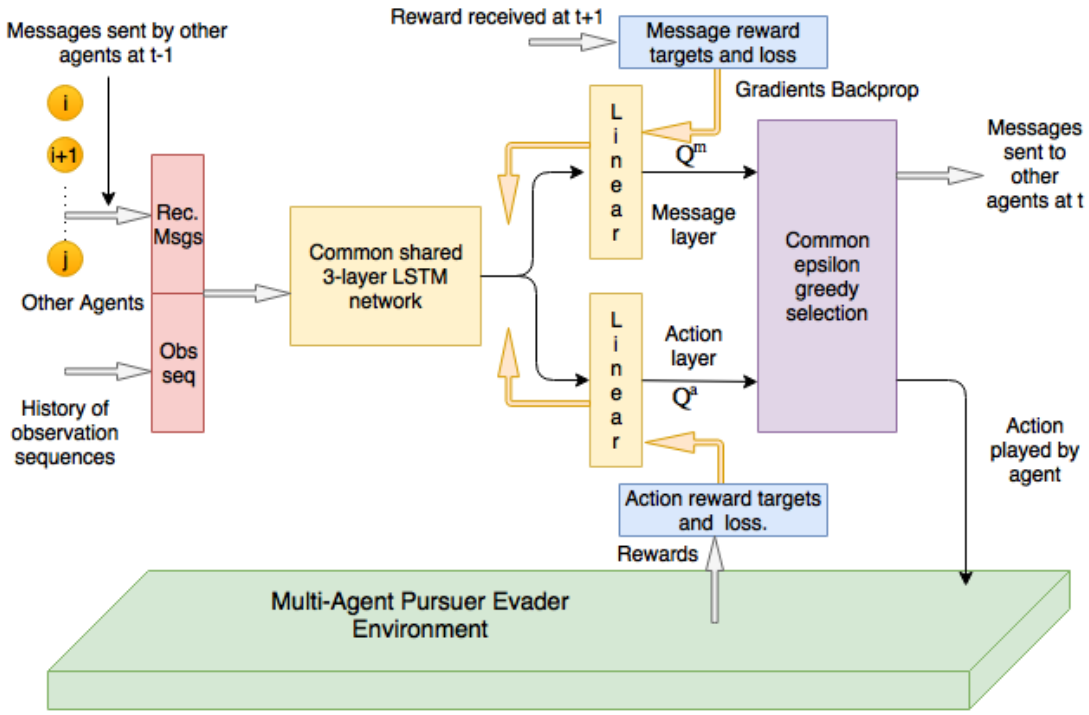


Figure 8.1: Architecture of the messaging algorithm

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.

$$\delta_i = \delta_i^a + \sum_{j \neq i} \delta_{ij}^m,$$

where

$$\delta_i^a = |Y_i^a - Q_i^a(O_t, m'_t, a_t)|,$$

$$Y_i^a = R_t + Q_{t_i}^a(O_{t+1}, m'_{t+1}, a''),$$

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

and

$$\delta_{ij}^m = |Y_{ij}^m - Q_{ij}^m(O_t, m'_t, m_t)|,$$

$$Y_{ij}^m = R_{t+1} + Q_{t_{ij}}^m(O_{t+1}, m'_{t+1}, m''),$$

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

where δ_i^a correspond to the TD-error of the action network and δ_{ij}^m 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

$$\beta\delta_i^a + (1-\beta)\sum_{j\neq i}\delta_{ij}^m,$$

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 3. The main architecture for the algorithm can be seen in Figure 8.1.

Algorithm 3 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**
$[O_k, r_k^a, a_k, O_{k+1}, m'_{k+1}, m_k, r_k^m] \leftarrow exp$

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

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

$DDQN^a_{gradient} \leftarrow$ Double-DQN gradient with target $r_k^a + \gamma R^a$
$DDQN^m_{gradient} \leftarrow$ 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 $T > T_{max}$

Chapter 9

Experiments and Results

9.1 Recent Algorithms on Atari Games

The trust region policy optimization algorithm does not provide better results than the others. For the implementation of the other two algorithms to make agents play Atari games we used Python with Tensorflow-Deep Learning Library. For the algorithms the input to the neural networks-the states were the four most recent screen frames of the games as explained before. For the Atari game simulators OpenAI gym python libraries were used which provide a simple way to manage the games and the interactions with the game.

The Deep Q-Learning algorithm was implemented with the architecture mentioned in [12]. The code was run on a server with a K40 GPU for the neural network training. But results obtained were not comparable to that of the paper's. This might be because the authors have trained the algorithm for a week whereas we were not able to access the GPU for such long periods of time. Even in the 1/2 days we could train continuously the progress in learning to get more rewards was not as expected.

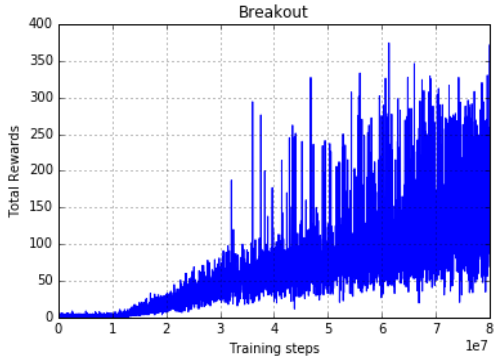
Asynchronous actor critic algorithm implementation : In this implementation, all threads performed updates after every 5 actions. The agents used a network which had a convolutional layer with 16 filters of size 8x8 with stride 4, followed by a convolutional layer with 32 filters of size 4x4 with stride 2, followed by a fully connected layer with 256 hidden units. All three hidden layers were

followed by a rectifier nonlinearity. In order to reduce the number of parameters the value network and the policy network shared these set of parameters. The output of this network is fed to a softmax layer corresponding to the number of actions for the policy network and to a linear layer for the value network. The state/the set of 4 images are the inputs to the network. Shared RMS prop was used for the stochastic updates (shared because the g parameter in RMS prop is also shared). A learning rate of 0.0007 was used which was linearly reduced to zero over the course of the training. Python has an issue of not supporting multi-threading meaning that only one thread handles the entire python process. To overcome this we had to use multi-processing where we spawn multiple processes instead of threads. But on doing this there is an issue of sharing parameters that arises as processes normally do not share memory like threads. To combat this we needed to use special multiprocessing shared arrays that only handle 1-D data and hence we stored the shared parameters in a 1-D array. So we had to shape and reshape the thread parameters everytime we needed to update the shared parameters. We used the algorithm on three games-Breakout, Pong and Beamrider. We used around 30 processes for Breakout and Beamrider while around 24 processes for Pong. We trained the network on each of them separately for more than a day each over about 80 million steps. Once fixed except for the number of actions for each game we did not change the hyperparameters between games. Due to the time taken to train these games we restricted to training and analyzing only three games.

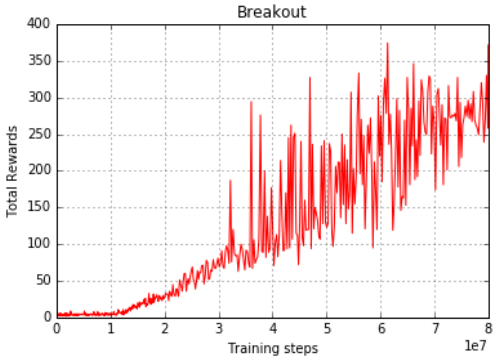
The results obtained on training with the Asynchronous Actor Critic algorithm for 80 million training steps were comparable to that in the paper [13]. They are shown in Figure 9.1. The agents trained were able to reach human level of play.

9.2 Weight Normalization

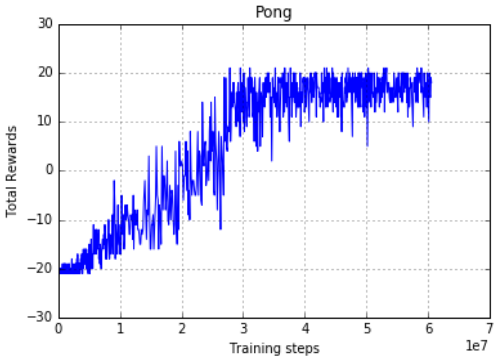
With the concept of weight normalization we were able to improve upon the Asynchronous Actor Critic algorithm. It improved the performance of the agents, mainly for the game of Breakout. The implementation and architecture is similar to the Asynchronous Actor Critic Algorithm discussed above but with the addition of weight normalization. The results we got on Breakout after training the games with this modified algorithm can be seen in Figure 9.2. In fact the new changed



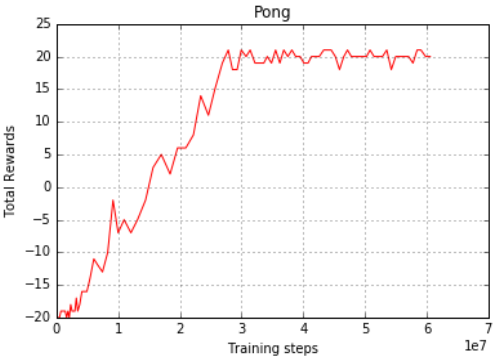
(a) Rewards collected for each episode



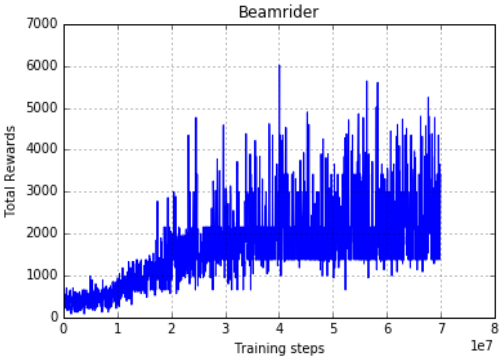
(b) Max of rewards collected over 10 episodes



(c) Rewards collected for each episode



(d) Max of rewards collected over 10 episodes



(e) Rewards collected for each episode

(f) Max of rewards collected over 10 episodes

Figure 9.1: Results obtained on training using the Asynchronous actor critic algorithm for 3 different games. It shows the scores at different training steps for the games.

algorithm is able to get rewards of more than 100 while the usual algorithm did not even reach rewards of 50. So the training speed is almost twice as fast after normalization.

Figure 9.2: Comparing the performance with weight normalization

9.3 Single-Agent Pursuit Evasion

By applying the methods discussed in Chapter 7 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.

Implementation: The implementation of the DNNs were similar to that done with the Asynchronous Actor Critic implementation, but without considering the threading issues as we used Deep Q-Learning. The neural network architecture for the algorithm can be seen in the Figure 7.1. The training was done using a NVIDIA Pascal Titan X GPU.

Since by our reward definition, we could not assess the performance of the agents with the total reward we used other parameters for performance analysis. So we noted the number of steps required to complete an episode over a batch of episodes and the total number of number episodes completed in the course of training, then used them to check the performance. In Figure 9.3 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

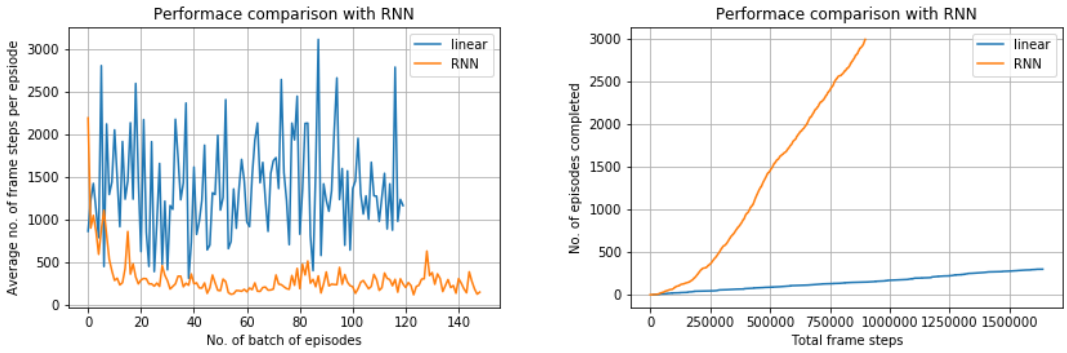


Figure 9.3: Performance using RNNs vs linear function approximator

than the linear function approximator case. So the agent is able to learn quickly and effectively with the help of the LSTMs.

9.4 Multi-Agent Pursuit Evasion

For the multi-agent pursuit evasion, 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 and an implementation for each pursuer similar to the single agent pursuer evader architecture and tuned some 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.

To check the performance of the messaging method we tried to train the agents using the independent learners [11] method, where each agent would simply ignore the other agents and follow their own separate reinforcement learning algorithms. We used the same parameters for measurement of performance as the single-agent case. From Figure 9.5 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 9.4. 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.

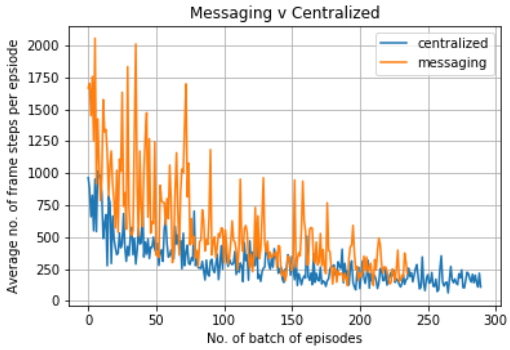


Figure 9.4: Comparing performance with centralized learning

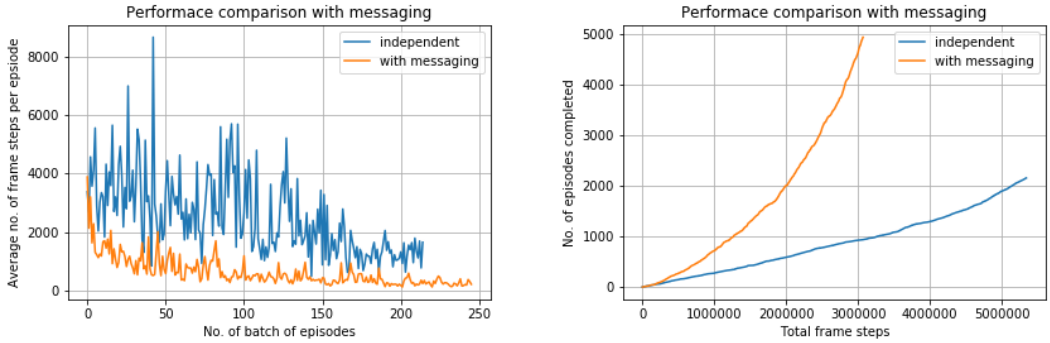


Figure 9.5: Performance comparison of messaging with DNNs and independent learners

Chapter 10

Conclusions and Future Work

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 a significant performance improvement using 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.

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.

Bibliography

[1] Michael Bowling and Manuela Veloso. Rational and convergent learning in stochastic games. In *International joint conference on artificial intelligence*, volume 17, pages 1021–1026. LAWRENCE ERLBAUM ASSOCIATES LTD, 2001. 22

[2] Lucian Busoniu, Robert Babuska, and Bart De Schutter. A comprehensive survey of multiagent reinforcement learning. *IEEE Transactions on Systems Man and Cybernetics Part C Applications and Reviews*, 38(2):156, 2008. 5, 22

[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*, pages 2137–2145, 2016. 23

[4] Matthew Hausknecht and Peter Stone. Deep recurrent q-learning for partially observable mdps. *arXiv preprint arXiv:1507.06527*, 2015. 20

[5] Nicolas Heess, Jonathan J Hunt, Timothy P Lillicrap, and David Silver. Memory-based control with recurrent neural networks. *arXiv preprint arXiv:1512.04455*, 2015. 7

[6] Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. *arXiv preprint arXiv:1502.03167*, 2015. 12

[7] Sham Kakade and John Langford. Approximately optimal approximate reinforcement learning. In *ICML*, volume 2, pages 267–274, 2002. 10

[8] Spiros Kapetanakis and Daniel Kudenko. Reinforcement learning of coordination in cooperative multi-agent systems. *AAAI/IAAI*, 2002:326–331. 22, 23

[9] Guy Lever. Deterministic policy gradient algorithms. 2014. 7

[10] 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*, pages 64–69. IEEE, 2007. 22

[11] Laetitia Matignon, Guillaume J Laurent, and Nadine Le Fort-Piat. Independent reinforcement learners in cooperative markov games: a survey regarding coordination problems. *The Knowledge Engineering Review*, 27(01):1–31, 2012. 32

[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. *Nature*, 518(7540):529–533, 2015. 7, 9, 11, 28

[13] 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. *arXiv preprint arXiv:1602.01783*, 2016. 8, 9, 11, 29

[14] Tim Salimans and Diederik P. Kingma. Weight normalization: A simple reparameterization to accelerate training of deep neural networks. *CoRR*, abs/1602.07868, 2016. 14

[15] Tom Schaul, John Quan, Ioannis Antonoglou, and David Silver. Prioritized experience replay. *arXiv preprint arXiv:1511.05952*, 2015. 19

[16] John Schulman, Sergey Levine, Philipp Moritz, Michael I. Jordan, and Pieter Abbeel. Trust region policy optimization. *CoRR*, abs/1502.05477, 2015. 9, 10

[17] Hado Van Hasselt, Arthur Guez, and David Silver. Deep reinforcement learning with double q-learning. In *AAAI*, pages 2094–2100, 2016. 20

[18] Ronald J Williams. Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Machine learning*, 8(3-4):229–256, 1992. 4