Deep Reinforcement Learning Algorithms and their Applications

M.E Project

Rohit R.R

Indian Institute of Science, Bangalore

Brief Recap

Deep Learning Effects on RL Algorithms

- Enables us to get general AI agents that can learn variety of tasks in different environments.
- With Deep Neural Networks like Convolutional Neural Networks we are able to learn complex representations of the real world through images.
- Memory property of Recurrent Neural Networks help to capture temporal aspects of data.

Deep Reinforcement Learning

Learning ability of Deep Neural Networks is used along with reinforcement learning algorithms to handle complex tasks in high dimensional state spaces.

General Notations

At each time t$\epsilon\{1,2,..\}$
State s$_t$ ϵS
Action a_t ϵA
Reward r_t ϵR
The environment's dynamics are characterized by
State transition probabilities
$P^a_{ss'} = P_r\{s_{t+1} = s' | s_t = s, a_t = a\}$
The total discounted reward we get over T time steps,
$R_t = \sum_{t'=t}^{T} \gamma^{t-1} r_t$
Q-value function,
$Q(s,a) = E\{R_T | s_t = s, a_t = a\} = R_{s,a}$
Policy,
$\pi(a|s) = P_r\{a_t = a | s_t = s\}$

Issues in using Deep Neural Networks

1. Unlike in supervised setting the "Data" we get to train the DNN is not IID ,so making DNNs to converge is a big challenge.
2. RL algorithms generally are "slow" iterative algorithms and addition of DNNs further increase computational requirements and the time to train, so scaling and training fast is difficult.

Atari Games

Deep Q Learning

Q-function is parameterized using a DNN to form the function approximator - $Q(s, a; w)$ [4]

Algorithm

For every step t,

1. ϵ-greedy policy
   Choose a random action with probability ϵ
   otherwise, choose action $= argmax_a Q(s, a)$
2. Store experiences in the replay memory.
3. Sample from the experience replay and generate,
   $$\text{Reward target} : r_{t'} + \gamma \left[\max_{a'} Q_{target}(s', a'; w^-) \right]$$
4. Policy Improvement step:
   $$L_i(w_i) = E_{s,a,r,s'} \{ (r_{t'} + \max_{a'} Q_{target}(s', a'; w_i^-) - Q(s, a; w_i))^2 \}$$
   $$\nabla_{w_i} L(w_i) = E_{s,a,r,s'} \{ (\text{Reward target} - Q(s, a; w_i)) \nabla_{w_i} Q(s, a; w_i) \}$$
   Use stochastic gradient descent to update the w parameters.

Weight Normalization

- Has shown improvements in training of networks for noisy data. So considered the **RL scenario to be a noisy one.**
- It reparameterizes the weights corresponding to the activation function input as,

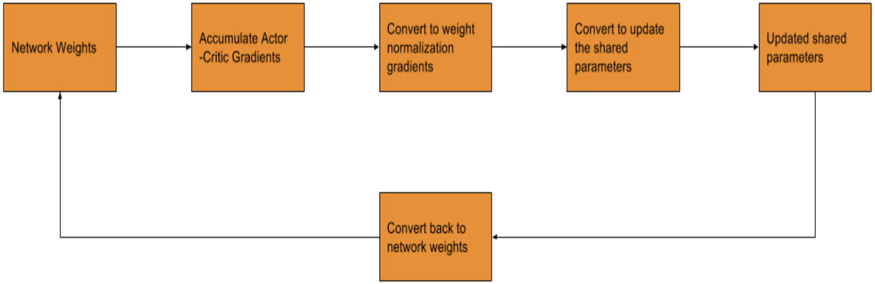
$$w = \frac{g}{||v||} v$$

- The extra variables do not increase computation costs much as the gradients can be calculated with the usual w gradients,

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

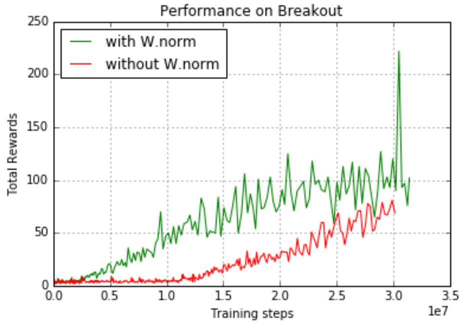
- It scales the gradient and projects the gradient vector away from the current weight vector.

Modified Algorithm


Network Weights
Accumulate Actor -Critic Gradients
Convert to weight normalization gradients
Convert to update the shared parameters
Updated shared parameters
Convert back to network weights

Experimental Results

- Improved upon the Asynchronous Actor Critic Algorithm.



Second Part

Application of Deep RL

Searched for a suitable application for these Deep RL algorithms. We chose pursuit evasion because,

- Extensive applications, such as searching buildings for intruders, traffc control, military strategy,and surgical operations
- Was feasible to create a good simulator
- Provided flexibility to test in complicated ways.

Pursuit-Evasion games

Pursuit-Evasion games are about guiding one or a group of pursuers to catch one or a group of moving evaders

Construction of Simulator and the Game

Used Python libraries PYMUNK and PYGAME to construct the simulator.

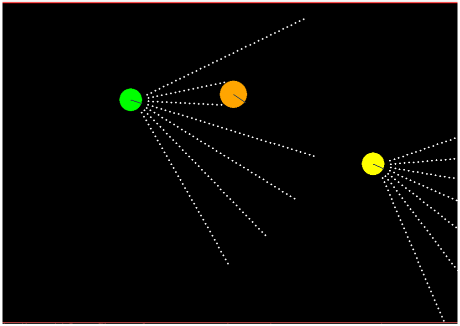


Figure: Screenshot of the simulator

Characteristics of the Game

- State/Observations : The sensor values with range upto 100 and it's own location.
- Actions : Each agent has about 3 actions - Rotate clockwise, anti-clockwise or hold it's direction and move forward.
- Reward : A reward of 100-500 if the agent captures the evader.
- Episode : Episode ends when the evader is caught.

Single Agent Pursuer Evader

One agent (pursuer) and one evader.

- The evader random case is not very complicated and can be handled with a simple linear function approximator.
- When the evader follows a specific path away from pursuer, the states do not reveal enough information.
- So we consider the game as a POMDP.

Other Issues

- There are long gaps with no rewards and the states do not change much.
- Plenty of experiences carry **no information**
- Sampling from experience replay randomly is inefficient.

We used some two main methods to handle these issues.

DQN with Priority Replay

- Want to sample **"important"** experiences more often [6].
- We consider experiences with larger **absolute TD-error** to be more important.
- $$\delta = [Y_t^{DDQN} - Q(S_t, A_t)]$$
  $$Y_t^{DDQN} = R(S_t, A_t) + Q_{target}(S_{t+1}, argmax_a(Q(S_{t+1}, a)))$$
  where Y_t^{DDQN} is the Double DQN target[1] .
- Helps in reducing overestimations.

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

DQN with Priority Replay (Contd...)

- If we prioritize greedily, transitions with initial low TD-error may not get replayed.
- Might overfit to certain experiences. To overcome such issues we use stochastic sampling.
- Experience j with TD-error δ_j, generate

$$P(j) = \frac{p_j^\alpha}{\sum_k p_k^\alpha}$$

- $p_j = |\delta_j|$ or $p_j = 1/rank(j)$ and $\alpha \in [0, 1]$, is the prioritization factor.

Deep Recurrent Q-Learning

- To handle the POMDP problem, we use recurrent neural networks [2].
- Use special kind of RNNs, LSTMs.

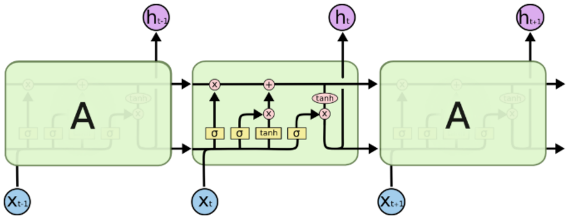


Figure: LSTM structure

- Use the LSTMs ability to learn Long term dependencies.

Deep Recurrent Q-Learning (Contd...)

- Input is a sequence of past observations O.
- The final layer is preceded by an LSTM to give $Q(O, a; \theta)$.
- A 16 layer LSTM was used.
- The hidden states of the LSTM need to be handled.
- Use Bootstrapped random updates.
- Reduces the gap between $Q(O, a; \theta)$ and $Q(S, a; \theta)$

Our Architecture

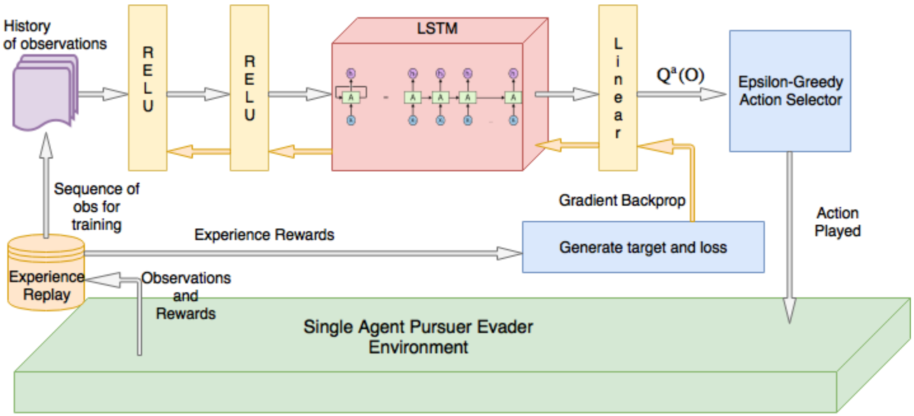
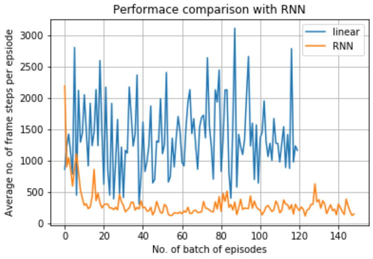


Figure: Architecture used for the Single Agent case

Results



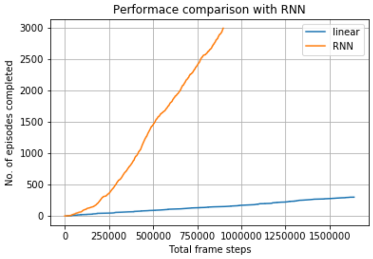


Figure: Training Performance using DRQN

Multi-Agent Reinforcement Learning

- Represented by $< X, U_1, U_2, ..., U_n, f, \rho_1, \rho_2, ..., \rho_n >$
- Joint action set:- $U = U_1 \times U_2.... \times U_n$
- State transition probability function :- $f : X \times U \times X \rightarrow [0, 1]$
- Reward functions:- $\rho_i : X \times U \times X \rightarrow \mathbb{R}, i = 1, 2, 3...., n$
- Policy functions $h_i : X \times U_i \rightarrow [0, 1]$
- Joint policy function, h
- Q-function, $Q_i^h : X \times U \rightarrow \mathbb{R}$ of each agent i,

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

Multi-Agent Reinforcement Learning

- In centralized learning, we solve a single MDP/POMDP over the joint actions and policies.
- In decentralized learning, each agent solves separate MDPs/POMDPs.
- Environment for each agent is non-Markovian.
- Difficult to get convergent algorithms in such cases.

Muti-Agent Pursuit Evasion

Game Features

- **Multiple Pursuers** chasing a single evader.
- Construction similar to single agent case.
- Evader moves away from the **closest pursuer** to itself.
- Pursuers **coordinate** with each other to capture the evader.

We concentrate more on the decentralized case.

Joint Action Learners with DQN

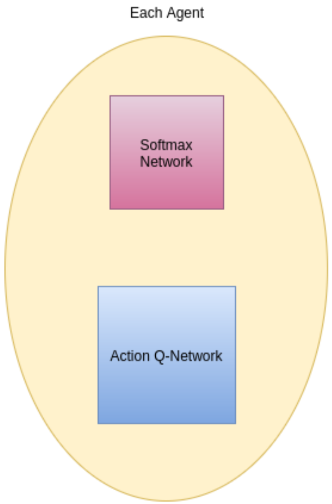
- Each agent have their own Q-network over states and all actions- $Q_i(x, a_1, a_2, ..., a_n)$
- Each agent follows own Q-learning algorithm over Q-values of it's own actions.
- In order to get such Q-values, the expected value $(EV)$is used.

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

  where $Pr^k(a^{-i}[k])$ is the probability of agent k choosing action $a^{-i}[k]$
- To get $Pr^k(a^{-i}[k])$, an empirical distribution of joint actions over each state is used.

Joint Action Learners with DQN (Contd ...)

- Due to large state sizes, we use a softmax network to approximate the distribution.
- We train the Softmax network simultaneously with the Q-network.
- Still the number of parameters to train were very high to get decent results.



Messaging with DQN

- Allow agents to communicate with each other through **small discrete messages**.
- Messages need to be ignored and interpreted properly, making it a complicated learning task.
- Each agent can transmit any of the M discrete messages - $\{1, 2, 3, .., M\}$.
- Sensor values along with the agents locations are the observations received by each agent.
- Use Deep Q-Learning algorithm to handle the messaging also.

Messaging with DQN (Contd...)

- Each agent i has message networks representing $Q_{ij}^{m}(O_t, m'_t, m_t)$: Q-value of sending message m_t from agent i to j, where
  O_t - Sequence of past observations received by the agent at step t,
  m'_t - Collection of all the messages sent to the agent which were generated in the previous time step.
- They also have the action Q-networks : $Q_i^a(O_t, m'_t, a_t)$
- Input is a concatenated version of the observation sequence and the received messages.

Messaging with DQN (Contd...)

- Messages are sent using Q_{ij}^{m} values and a message selector.
- Effects of current messages sent is visible only after one time step.
- To get the message rewards at step t, we wait for the rewards at $t+1$, R_{t+1}.
- Action rewards at step t is the usual R_t.
- Use priority replay
- But many Q-values associated with each experience.

Messaging between agents - Priority Replay

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.

Messaging between agents - Priority Replay Variant

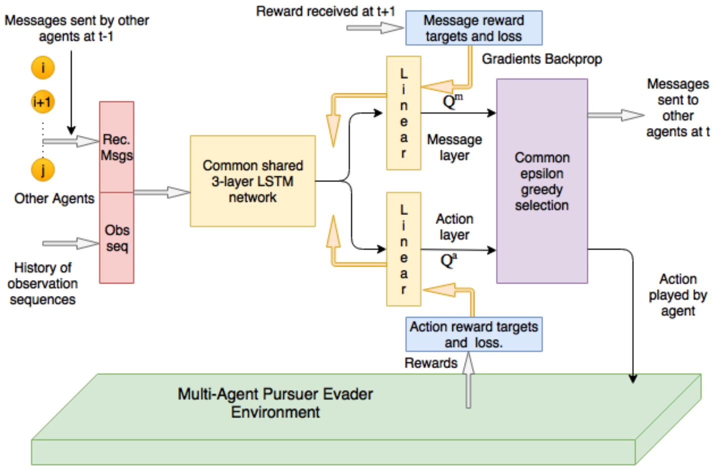
As a variant, we update the total TD-error as

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

where $\beta \in [0,1]$, is a hyper parameter.

- β can be adjusted in order to make either the action network or the message network more sensitive.
- If β is high, experiences with larger action TD-errors would be considered more important.
- Helps in reducing unnecessary noise while training.

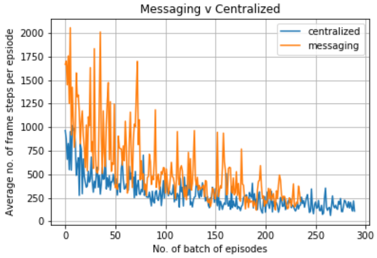
Our Architecture

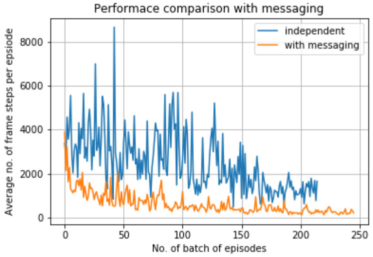


Messages sent by other agents at t-1
i
i+1
j
Other Agents
Rec. Msgs
Obs seq
History of observation sequences
Common shared 3-layer LSTM network
Reward received at t+1
Message reward targets and loss
Gradients Backprop
Linear
Q^m
Message layer
Linear
Action layer
Q^a
Common epsilon greedy selection
Messages sent to other agents at t
Action played by agent
Action reward targets and loss.
Rewards
Multi-Agent Pursuer Evader Environment

Figure: Architecture used for the Multi-Agent case

Results





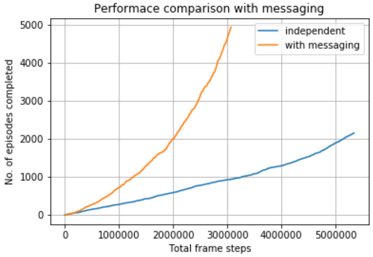


Figure: Training results with the messaging algorithm

Conclusion

- We were able to play Atari Games which were infeasible with prior reinforcement learning algorithms.
- Improved upon the Asynchronous Actor Critic algorithm with weight normalization.
- Able to handle problems like partial observability to train agents in Single Agent Pursuit Evasion.
- Allowing multiple pursuers to message with discrete symbols using DNNs, we are able to make them coordinate and capture the evader
- Agents learn to perform better than simply being independent learners.

Future Work

- 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, we might need to use non-cooperative multi-agent reinforcement learning algorithms.
- In our Pursuer Evader application we did not vary the speeds of the agents. So the acceleration of the pursuer can also be controlled as an action apart from controlling the direction of movement of the pursuer. We can train the agent to increase their speed while chasing the evader and move at slower speeds while simply exploring.

References I

[1] 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.

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

[3] 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).

References II

[4] 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.

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

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

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