Exploring Reinforcement Learning on various Environments

Sri Amarnath Mutyala
University at Buffalo
Buffalo, NY
sriamarn@buffalo.edu

Nandini Chinta
University at Buffalo
Buffalo, NY
nandinic@buffalo.edu

Venkata Krishna Boinepally
University at Buffalo
Buffalo, NY
vboinepa@buffalo.edu

Abstract

In this report, we applied Reinforcement learning algorithms to solve different environments. Specifically, we first explored value iteration algorithms like Monte Carlo Control, Q-Learning and SARSA. Later, we explored some Deep Q Networks on OpenAI LunarLander-V2. We also extend DQN to Multi-agent environments and explore the concepts of fairness in MARL.

The code for this project is located at https://github.com/artiemar/CSE-676-Final

Initially, another topic was chosen. But after discussing with Professor, we have chosen this as our final project.

1 Partially observable Markov decision process (POMDP)

Markov decision process is a flexible mathematical framework to model sequential learning problems. We express transitions in the environment as moving between states by taking actions and receiving a reward signal from the environment.

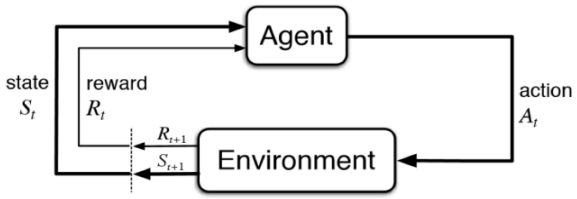


Figure 1: Markov decision process model

Our objective is to find a policy π^* that maximizes the cumulative discounted reward $\sum_{t\geq 0}^{t} \gamma^t r_t$
In partially observable MDP's the agent only receives observations, reward signal from the environment. The whole environment is not visible to it.

1.1 Value Iteration

In Value Iteration we assign value to each state and update them based on the Bellman update.

In this problem, we take the environment of Snake and Ladders board game and design an algorithm using Q Learning and On Policy Monte Carlo and TD0 SARSA.

1.2 Environment

We simulate a grid world based on the famous Indian game of Snake and Ladders. Where two players start the left bottom corner of the board i.e., state 0 and the objective is to reach the goal at state 100. An extensive discussion of formulating the game of snake and ladders as an MDP is found in the references below.

This is a discrete action and observation space.

Actions: Move {1,2,3,4,5,6} positions. (Based on rolling the dice but the agent can choose its actions here).

States: [0,100] discrete mutually exclusive states.

Rewards: -1 for all states other than the final, +1 for state the final state.

Goal: - ***Learn to reach goal in least number of steps i.e., to learn the given board.***

1.3 Algorithm

We use epsilon decay strategy with a step function to handle the exploration and exploitation transition with epochs. We reset the test environment in the beginning and train on both On Policy Monte Carlo and Q Learning, SARSA Agents and compare the results at the end.

1.4 Results

We compare the results from the two agents by plotting the graph below.

The game has two modes being Deterministic and Stochastic. Since in MDPs we usually use the concept of transition probability between states. We wanted to introduce this behavior of stochasticity into the map.

In case of the stochastic mode of the game, the agent's action is reflected on the environment with a ***transition probability of 0.76***. We simulate this by transitioning with a random action b/w 1-6. We see good performance of the agent in stochastic and deterministic environments.
We see more variance in Monte Carlo Control, and we see that SARSA, Q-Learning both have low variance and a faster convergence.

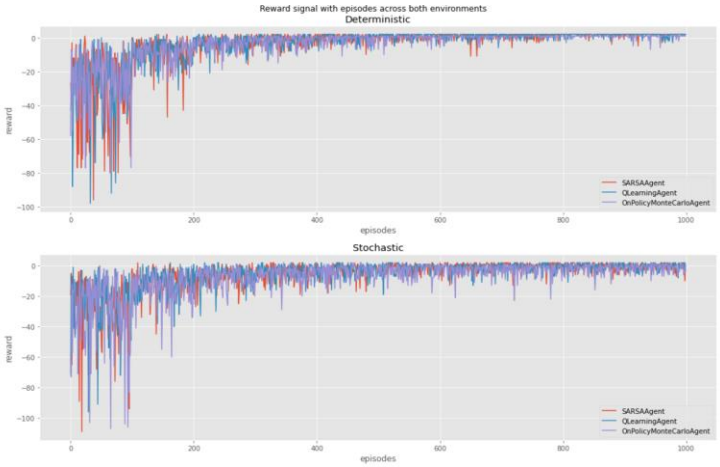
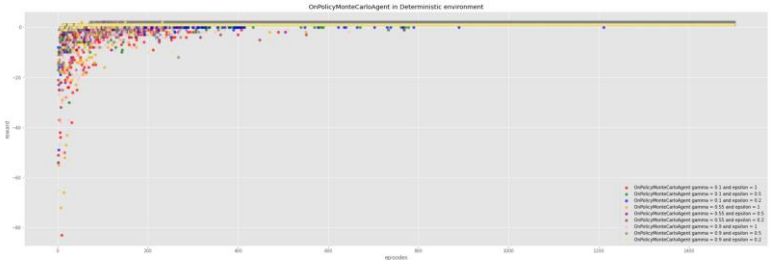


Figure 2: Comparison of rewards of Three Agents vs Episodes

We also perform some hyperparameter tuning by changing the hyperparameters.



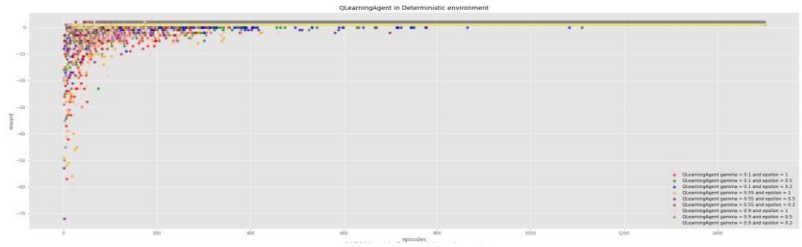


Figure 3: Comparison of rewards of two agents vs episodes in different hyperparameters

1.4 Summary

We were able to solve this environment using Q-Learning, SARSA, MonteCarlo control agents and compared their performance on both stochastic and deterministic environments.

2 Deep Q-Networks

Tabular value iteration methods like Deep Q-Networks don't scale well for large environments. So, for the purpose of value approximation, we can employ a neural network since they can approximate functions of high dimensions according to the Universal Approximation Theorem.

For this module, we choose to solve the OpenAI LunarLander-v2 environment. We use DQN and a variant called Double DQN to solve this environment. We also use experience replay and a target network with the DQN algorithm as first introduced in the Deepmind paper in 2017.

2.1 Problem Description

Our goal is to design a neural network that can take in the current state as input and give us value vector with dimension equal to actions.

2.2 Environment

This environment simulates the landing a space craft on the lunar surface. There are three main engines on the bottom, both the sides of the spacecraft.
This environment has discrete actions: Each engine can be *On* or *Off*.

The goal is to land at the origin (0,0). The state vector is the co-ordinate position of the spacecraft in that frame. Fuel is infinite but each firing is penalized with a goal to help learn quick landings.

The environment we used is of a discrete action space, since DQN works only on discrete action spaces.

States: The state one-hot encoding vector S consists of 8 dimensions. Cartesian coordinates, Linear Velocities, pose, angular velocities and Two Boolean variables to indicate if the right and left legs are in contact with the ground.

Action space: The action vector is discrete and consists of 4 actions. Do nothing, Firing left/right/main engines.

Rewards: -

Multiple reward signals:

1. Initial position to landing: +100 to +140
2. Moving away from the landing pad, leads to negative reward
3. Crashing leads to termination and a reward of -100.

4. Each leg in contact with the ground is given +10 reward.
5. Firing the main engine is -0.3 points for each frame and the side engines a penalty of -0.03 points. (Penalize long firing schemes)
6. Correctly landing at the origin is +200 reward.

2.3 Algorithm

Algorithm

Algorithm 1: deep Q-learning with experience replay.

Initialize replay memory D to capacity N
Initialize action-value function Q with random weights θ
Initialize target action-value function $\hat{Q}$ with weights $\theta^{-} = \theta$
For episode = 1, M **do**
  Initialize sequence $s_1 = \{x_1\}$ and preprocessed sequence $\phi_1 = \phi(s_1)$
  For $t = 1,T$ **do**
    With probability ε select a random action a_t
    otherwise select $a_t = \text{argmax}_a Q(\phi(s_t), a; \theta)$
    Execute action a_t in emulator and observe reward r_t and image x_{t+1}
    Set $s_{t+1} = s_t, a_t, x_{t+1}$ and preprocess $\phi_{t+1} = \phi(s_{t+1})$
    Store transition $(\phi_t, a_t, r_t, \phi_{t+1})$ in D
    Sample random minibatch of transitions $(\phi_j, a_j, r_j, \phi_{j+1})$ from D
    Set $y_j = \begin{cases} r_j & \text{if episode terminates at step j+1} \\ r_j + \gamma \max_{a'} \hat{Q}(\phi_{j+1}, a'; \theta^{-}) & \text{otherwise} \end{cases}$
    Perform a gradient descent step on $\left(y_j - Q(\phi_j, a_j; \theta)\right)^2$ with respect to the network parameters θ
    Every C steps reset $\hat{Q} = Q$
  End For
End For


```
lunar_lander_hyperparameters  = {
    'state_dim': env.observation_space.shape[0],
    'action_dim': env.action_space.n,
    'layer_dim': [150, 120],
    'lr': 0.001,
    'episodes': 10,
    'gamma': 0.99,
    'replay_batch_size': 64,
    'epsilon': 1,
    'epsilon_min': 0.01,
    'replay_buffer_size': 100000,
    'update_target_every': 4,
    'goal_threshold_score': env.spec.reward_threshold,
    'goal_threshold_episodes': 100,
}
```

```
Summary of Layers in the employed network:
Sequential(
  (Input layer): Linear(in_features=8, out_features=150, bias=True)
  (ReLU 0): ReLU()
  (Hidden layer 1): Linear(in_features=150, out_features=120, bias=True)
  (ReLU 1): ReLU()
  (Output layer): Linear(in_features=120, out_features=4, bias=True)
)
```

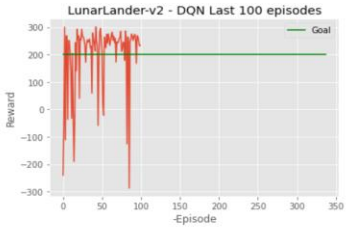
2.4 Double DQN

In the case of Double DQN we have two separate networks. Here we use the target network from earlier DQN implementation as the second network. We use one network to choose actions and the other to estimate the Q-value.

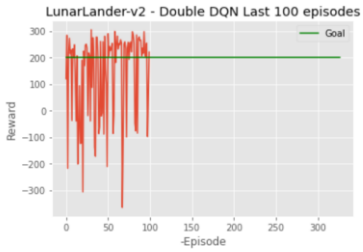
$$Q^*(s_t, a_t) \approx r_t + \gamma Q\left(s_{t+1} \cdot argmax_{a'} Q'(s_t, a_t)\right)$$

2.5 Results

We see good convergence for the goal of +200 rewards for 10 consecutive training epochs. The results are as below.




```
Epoch: 338      Total Reward: 300.3512396455469      Epsilon: 0.020417379446695434
Goal Reached!!
```




```
Epoch: 327      Total Reward: 237.14957094101064      Epsilon: 0.02317394649968494
```

2.5 Summary

We see that our implementation was able to solve the environment, but we see a lot of Variance in the total reward per epoch. To reduce the Variance, we should try to increase the batch size and we can try reducing the α, which is the learning rate.

3 DQN with CNN's

In the previous section, we have implemented DQN, but we wanted to extend the network to use Convolution layers and input the environment renders (frames) as input to the network. As mentioned in DeepMind paper [5] wherein they implemented a buffer of n frames for the network to learn the velocity and general direction of the player in games like Atari Breakout. We wanted to apply similar technique to the problem. We faced compatibility issues when we extended our DQN implementation, so we used an external implementation to simulate for this environment.

3.1 Code

```
QNetwork(
  (cnn): Sequential(
    (0): Conv2d(1, 32, kernel_size=(8, 8), stride=(4, 4))
    (1): ReLU()
    (2): Conv2d(32, 64, kernel_size=(4, 4), stride=(2, 2))
    (3): ReLU()
    (4): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1))
    (5): ReLU()
  )
  (fully_connected): Sequential(
    (0): Linear(in_features=3136, out_features=512, bias=True)
    (1): ReLU()
    (2): Linear(in_features=512, out_features=4, bias=True)
  )
)
```

3.2 Results

We couldn't obtain good results after training the agent as LunarLander-v2 is not well suited to solve using CNN based DQNs. We may obtain good results after hyperparameter tuning and a very long training time. We couldn't train it for that long as we lack the computational resources to do so.

4 Policy Gradient: Actor-critic models

In this environment, our task is to land a Lunar Lander on the ground at the designated flags by controlling the engine on or off.

4.1 Environment

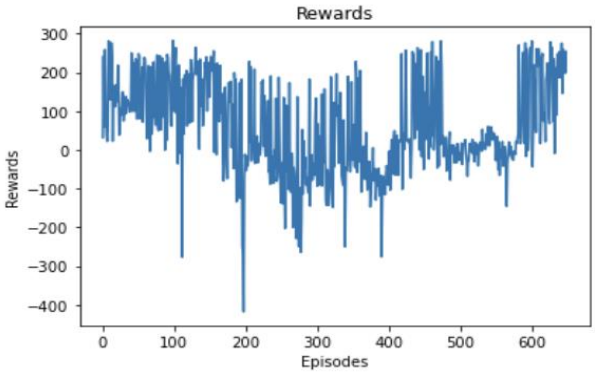
The observation space consists of 8 states, the current coordinates, the linear and angular velocity in each direction, the angle of tilt and Boolean to indicate whether it touched the ground or not. The action space has 4 actions, do nothing, fire left, fire right or fire main engines. The reward is in the range of 100-140 when coming down from top to bottom, if lander crashes, then reward of -100. Each leg contact is 10 points and 100 points on rest. Firing has some negative rewards with the intention of keeping it moves with minimum steps.

4.2 Code

In this Project, we have considered three hidden layers neural networks for both Actor and Critic networks with four nodes of output for actor and one state value as output for Critic. The actor is approximated to choose best action while Critic evaluates the action. We can say that Actor Critic is just a DQN with a policy gradient network to choose action as policy instead of epsilon-greedy policy where we use it in Q-Learning or DQN. We are here approximating the policy also using a network and updating the weights of the policy for every episode. And we are updating the weights of the critic for every timestep with a frozen target improvisation. We have used MSE (Mean Squared Error) loss and Adam optimizer for both the actor and the critic. SoftMax activation function for Actor to get the probabilities of all actions to choose from.

4.3 Results

With a hyperparameters set of 0.001 learning for both actor and the critic networks and 0.99 discount factor, three hidden layers of 128-512 nodes range, we get the following results.



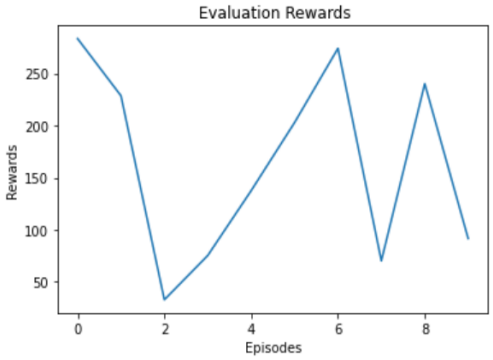


Fig 4.3.1: Rewards per episode (Training) Fig 4.3.2: Rewards per episode (Evaluation)

4.4 Summary

We can say that the agent started learning the environment after 300 episodes of training and converged at 600 episodes. We still see high variance in the total rewards as we saw earlier in the DQN and DDQN implementations. Also, our implementation did not perform well as we in see the evaluation results. It fails short of the required goal of 200 total reward per episode.

5 Extending to Multi-Agent Environments

Multi-Agent Reinforcement Learning is a subfield of Reinforcement Learning with multiple agents acting on an environment. Since there are multiple agents. Most multi-player games can be thought of as multi-agent interactions. Multi agent environments can be modelled as POMDPs we discussed above.

5.1 Problem Description

Fairness in Multi-Agent Reinforcement Learning is how agents are going to cooperate with each other maximizing not only their own future cumulative rewards but also considering the other agents' situation.

5.2 Environment

Environment implemented here is a custom environment with three agents having a common subgoal and independent final goals where they need to share the resources from the subgoal and reach the final goal within the limited timesteps. Fairness here is defined by the limited time given to each agent to reach the goal, limited resources to share and individual priorities for each agent.

5.3 Code

We have implemented value iterations and Deep Q-networks on the environment. For an off-policy Q-Learning algorithm, each agent will have a separate Q-table to learn and separate Q-network for on-policy Deep Q-Networks (DQN) and Double DQN. Q-Learning gives the best action to take for a given situation. Using the epsilon greedy policy, we make the agent explore and exploit in a balanced way to learn more from the environment. In DQN we use neural networks that predict the best action to take given state. We update the weights of neural networks for best approximation of actions accordingly. Double DQN is where we work with two neural networks, one to predict the action and the other to evaluate the first network.
In this project we used a network of two hidden layers of 64 nodes for DQN and two hidden layers of 128 nodes for DDQN for each agent. We used Mean Squared Error loss to compute loss for the networks and Adam optimizer for backpropagation.

5.4 Results

Below are the graphs of each agent compared, for all algorithms we implemented.

5.4.1 Results – Q-Learning

With hyperparameters of 0.01 learning rate, 0.99 discount factor for future rewards the agents got converged at following episodes.

Fig 6.4.1.1 : Total Rewards per episode (Training)

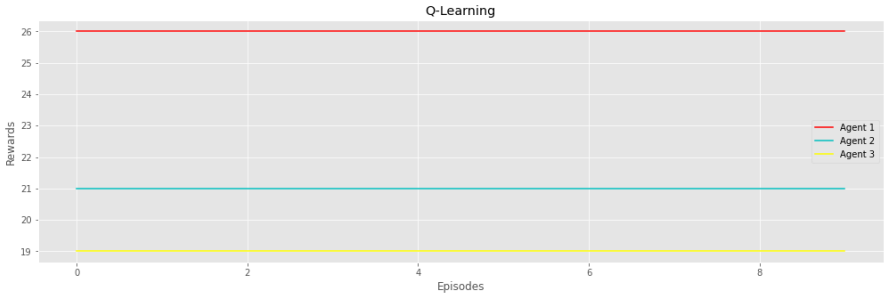


Fig 5.4.1.2 : Total Rewards per episode (Evaluation)

5.4.2 Results – Deep Q-Networks

With hyperparameters of 0.01 learning, 0.99 discount factor for future rewards, a network of two hidden layers with 64 nodes, the agents got converged at following episodes.

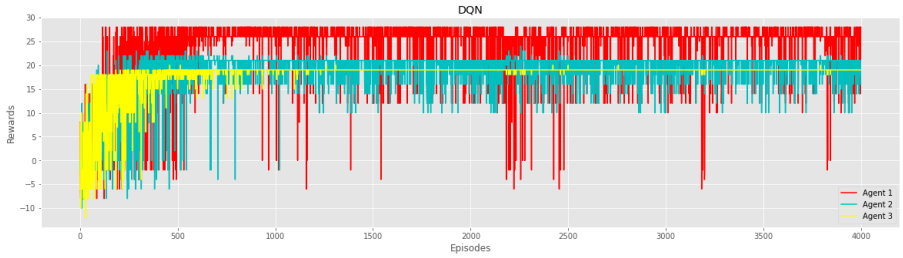


Fig 5.4.2.1 : Total Rewards per episode (Training)

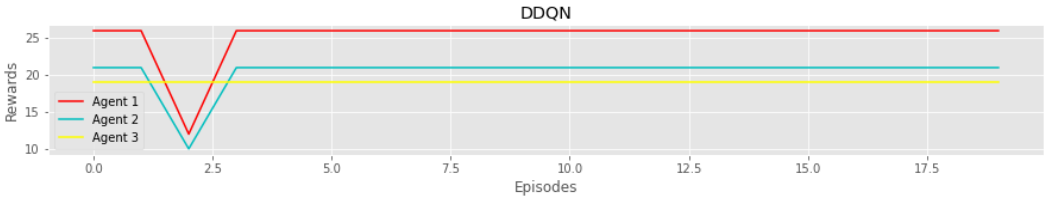


Fig 5.4.2.2 : Total Rewards per episode (Evaluation)

5.4.3 Results – Double Deep Q-Networks

With hyperparameters of 0.01 learning, 0.99 discount factor for future rewards, a network of two hidden layers with 64 nodes, the agents converged at following episodes.

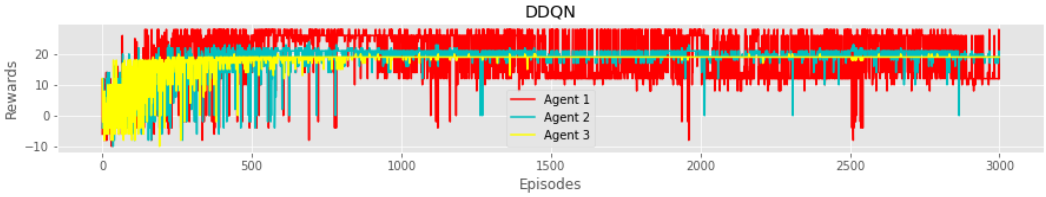


Fig 5.4.3.1: Total Rewards per episode (Training)

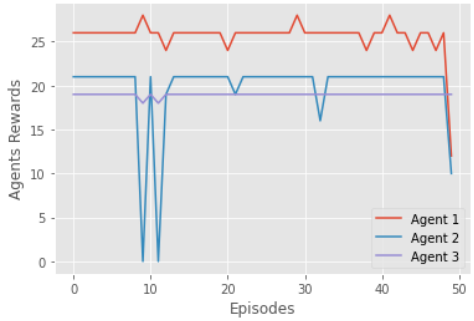


Fig 5.4.3.2: Total Rewards per episode (Testing)

5.5 Summary

We considered each agent's total rewards per episode as comparison metric to compare with other agents. Double DQN performed well when compared with DQN and Q-Learning. We can see that each agent is trained to retrieve maximum rewards possible in shortest path.

6 Unfinished Work

We wanted to explore more complex settings such as the real-world environments of improving fleet scheduling as defined in [7], other interesting topics as Parameter Sharing.

6.1 Fleet Scheduling

Fleet scheduling [7] is a complex high dimensional optimization problem for which there doesn't exist many heuristic algorithms. To solve this problem, we can model this problem as a RL problem of making sequential decisions. Though the environment looked interesting, when we tried to migrating the code to torch we faced some issues and the lack of dataset from Didi which was used in the referenced paper, made it hard to work on this problem.

6.2 Parameter Sharing

Parameter sharing was discussed in class as a means of reducing memory footprint and speeding up training. In multi agent scenarios, especially while training similar agents (as opposed to adversarial agents), we can speed up training by tying up the parameters of the networks of both the agents.

$$\Omega(w^A, w^B) = |w^A - w^B|^2$$

6.3 RLLib Baseline comparisions

While implementing the above algorithms we felt that it's better to compare the performance of our implementations with some baseline implementations for efficiency, variance, bias and generalizability. To this end we implemented a few baseline comparisions using an open source RL library called RLLib.

7 Conclusion

In this project we explored concepts in Deep Reinforcement Learning discussed in class by Professor. We implemented the tabular value iteration methods and more recent DQN and its variants. We also experimented with Conv-DQN on LunarLander-V2 environment, but we saw that it didn't perform as well as expected on this environment. We checked some basic multi-agent environments and explored some concepts related to fairness, global cumulative reward maximization etc... in preparation for the future work on Fleet Scheduling environment and most recent techniques like Parameter Sharing between cooperative agents.

8 References

[1] Professor's Lectures & Slides

[2] Rich Sutton's Home Page (incompleteideas.net)

[3] Simulating Chutes & Ladders in Python | Pythonic Perambulations (jakevdp.github.io)

[4] Gym (openai.com)

[5] Playing Atari with Deep Reinforcement Learning (deepmind.com)

[6] DQN with CNN: Recreating the Google DeepMind Network - DataHubbs

[7] [1802.06444] Efficient Collaborative Multi-Agent Deep Reinforcement Learning for Large-Scale Fleet Management (arxiv.org)

[8] RLlib: Industry-Grade Reinforcement Learning — Ray 1.12.1

[9] 10493aa88605cad5ab4752b04a63d172-Paper.pdf (neurips.cc)