Exploring Actor Critic Algorithms

Sudhir Yarram Krishna   Vamsi Velivela

50305566   50318486

1 Introduction

In this report we do a thorough analysis of various deep reinforcement learning algorithms to solve cartpole and Lunar Lander. In addition, we have also implemented PPO in image-based SpaceInvader-v2.

Algorithms

DQN

Deep Q-Network is one of the effective ways to train the reinforcement learning agent. It is highly effective when there are too many states and too many actions. It predicts the action on unseen state by the experience gained from the seen state using the Deep learning techniques. I implemented DQN, Double DQN and Dueling DQN on 6*6 Grid World,Cartpole and LunarLander.

Double DQN

The Double DQN is similar to Double Q-Learning. It has two Q estimators, Q1 estimates the best actions and Q2 evaluates Q1 for the selected action.

$$Q_1(s,a) \leftarrow Q_1(s,a) + \alpha(\text{Target} - Q_1(s,a))$$

$$\textbf{Q Target:} \quad r(s,a) + \gamma \max_{a'} Q_1(s',a')$$

$$\textbf{Double Q Target:} \quad r(s,a) + \gamma Q_2(s', \arg\max_{a'}(Q_1(s',a')))$$

Actor-Critic

Actor-Critic methods are temporal difference (TD) learning methods that represent the policy function independent of the value function.In the Actor-Critic method, the policy is referred to as the actor that proposes a set of possible actions given a state, and the estimated value function is referred to as the critic, which evaluates actions taken by the actor based on the given policy. We implemented TD Actor Critic algorithm on 6*6 grid world, Cartpole and LunarLander.

$$Q_w(s,a) \approx Q_{\pi_\theta}(s,a)$$

parameters:- Actor-critic algorithms maintain two sets of parameters.

1. Critic:- Updates action-value function parameters w. The critic is solving a familiar problem of policy evaluation.
2. Actor :- Actor Updates policy parameters θ, in direction suggested by critic.

Actor-critic algorithms follow an approximate policy gradient

$$\nabla_\theta J(\theta) \approx E_{\pi_\theta}[\nabla_\theta \log \pi_\theta(s,a) Q_w(s,a)]$$
$$\Delta\theta = \alpha \nabla_\theta \log \pi_\theta(s,a) Q_w(s,a)$$

Unlike value based algorithms, there is no need to store the previous computed values. It is significantly faster than the value based algorithm.

Reinforce

There are two broad ways of solving a reinforcement learning problem – They are calculating values functions like state values or Q-values by either approximating them or maintaining a tabular method. In this method, we approximate the state or action value to get the action and then use methods like epsilon decay to improve the policy. The second method is to approximate the policy directly.

$$\pi_\theta(s,a) = P[a|s,\theta]$$

These are called policy gradient methods. They target at modelling and optimizing the policy directly. We perform gradient ascent here to maximize performance.

$$\theta_{t+1} = \theta_t + \alpha \nabla J(\theta_t)$$

In policy gradient we get the probability of each action and then choose action randomly. The outputs are calculated using the sigmoid activation function.

$$\pi(a|s,\theta) = \frac{e^{h(s,a,\theta)}}{\sum_b e^{h(s,b,\theta)}}$$

The finally gradient of J is written as

$$\nabla_\theta J(\theta) = \mathbb{E}_\pi[Q^\pi(s, a)\nabla_\theta \ln \pi_\theta(a|s)]$$

We use the log likelihood function to calculate the loss. Which is multiplied with the Return that we calculate this return can either be the return of the entire episode (MC policy gradient) or the Q-value calculated by the critic (1 Step TD Actor-Critic).

Reinforce is also called as Monte-Carlo Policy Gradient. In this, we calculate the total discounted return of the entire episode to calculate the gradient.

The update function looks like

$$\Delta\theta_t = \alpha G_t \nabla_\theta \log \pi_\theta(s_t, a_t)$$

The choose action function takes the probabilities from the model and chooses an action based on the probabilities.

The learn function calculates the discounted sum of the rewards for the entire episode. We use a baseline here that calculates the mean of the all rewards and standard deviation of all the rewards and calculates sum-mean/standard deviation. And train the model based on this baseline and states of the entire episode.

PPO

In reinforce and other actor critic algorithm we still face a lot of variance in the policy and drastic changes in the policy. The agent needs to take importance sampling to reduce this variance. One way to do this is by clipping the ration of the new policy and old policy by a certain factor. TRPO uses KL divergence to measure the difference between the two new policies.

In PPO with clipped objectives, we maintain two networks, one with the current policy that we want to refine and second that we use to collect samples.

$$\pi_\theta(a_t|s_t) \qquad\qquad \pi_{\theta_k}(a_t|s_t)$$

With the idea of importance sampling, we evaluate a policy samples collected from older policy.

$$\underset{\theta}{\text{maximize}} \quad \hat{\mathbb{E}}_t\left[\frac{\pi_\theta(a_t \mid s_t)}{\pi_{\theta_{\text{old}}}(a_t \mid s_t)}\hat{A}_t\right]$$

We synchronize the second network to the first network using a soft update. With clipped objective, we calculate the ratio between the new policy and old policy. Our objective function becomes:

$$\mathcal{L}_{\theta_k}^{CLIP}(\theta) = \mathop{\mathrm{E}}_{\tau \sim \pi_k}\left[\sum_{t=0}^{T}\left[\min(r_t(\theta)\hat{A}_t^{\pi_k}, \mathrm{clip}\left(r_t(\theta), 1-\epsilon, 1+\epsilon\right)\hat{A}_t^{\pi_k})\right]\right]$$

We are clipping the ration of the old and new policy between 1-€ and 1+€ generally € is set to 0.1 or 0.2 in the PPO paper. This is very simple to implement.

Environments

LunarLander-v2

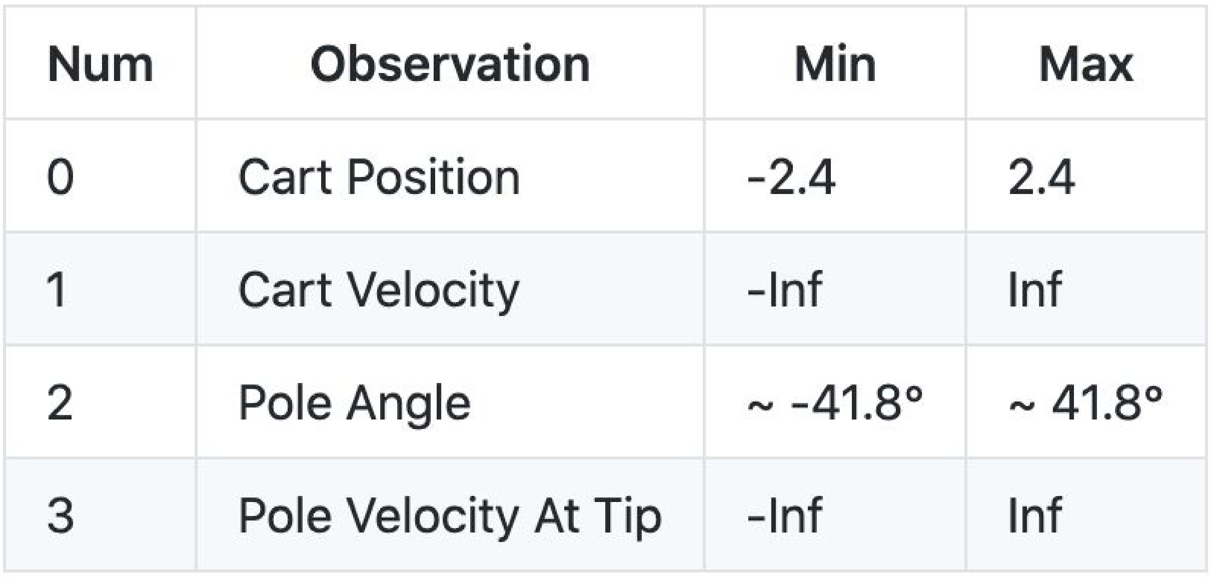
Landing pad is always at coordinates (0, 0). Coordinates are the first two numbers in state 157 vector. Reward for moving from the top of the screen to landing pad and zero speed is 158 about 100...140 points. If lander moves away from landing pad it loses reward back. 159 Episode finishes if the lander crashes or comes to rest, receiving additional -100 or +100 160 points. Each leg ground contact is +10. Firing main engine is -0.3 points each frame. 161 Solved is 200 points. Landing outside landing pad is possible. Fuel is infinite, so an agent 162 can learn to fly and then land on its first attempt. Four discrete actions available: do 163 nothing, fire left orientation engine, fire main engine, and fire right orientation engine.

Possible Actions

Actions here is four floats [main engine, left-right engines, fire main engine, and fire 168 right orientation engine]

CartPole

Cartpole has two actions 0 to push left and 1 to push right. It gets a reward of 1 for every step taken, including the termination step. The threshold is 475. The objective is to make the pole stand for as long as possible.



Num	Observation	Min	Max
0	Cart Position	-2.4	2.4
1	Cart Velocity	-Inf	Inf
2	Pole Angle	~ -41.8°	~ 41.8°
3	Pole Velocity At Tip	-Inf	Inf

SpaceInvader-v4

We consider the task of learning to play Space Invaders, a popular Atari console game. Formally, an agent interacts with an environment E in a sequence of actions, observations and rewards. In our case the environment is the Atari emulator for Space Invaders. The game play is discretized into time-steps and at each time step, the agent chooses an action from the set of possible actions for each state A = {1, 2, ...L}. The emulator applies the action to the current state, and brings the game to a new state, as shown in figure 1. The game score is updated and a reward is returned to the agent. We formalize this problem as follows

- State s: A sequence of observations, where an observation, depending on the state space we are operating in, is a matrix representing the image frame or a vector representing the emulator's RAM state.
- Action a: An integer in the range of [1, L]. In case of Space Invaders L = 6 and the actions are {FIRE (shoot without moving), RIGHT (move right), LEFT (move left), RIGHTFIRE (shoot and move right), LEFTFIRE (shoot and move left), NOOP (no operation)}.
- Reward r: Reward returned by the environment, clipped to be in the range [−1, 1]

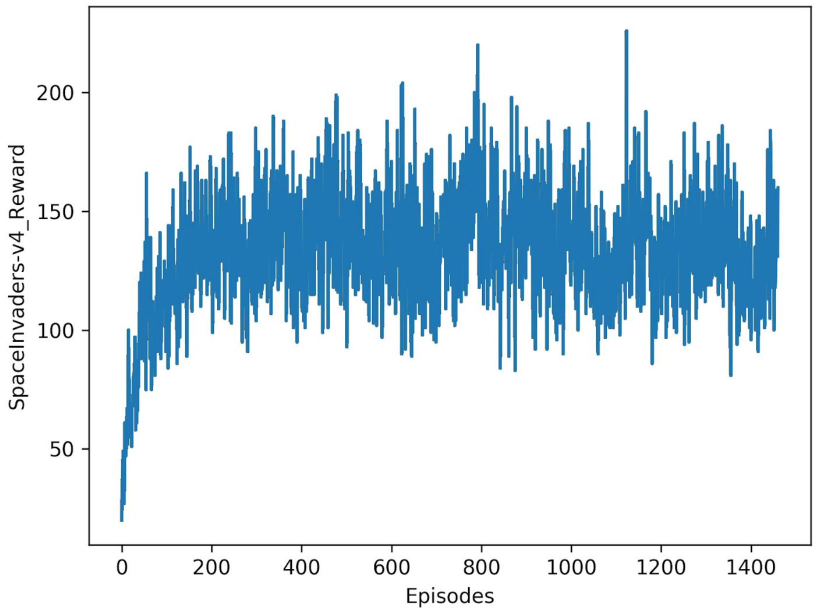
2 Results

2.1 Space Invaders

We use the following architecture for extracting features and to output actions and value.

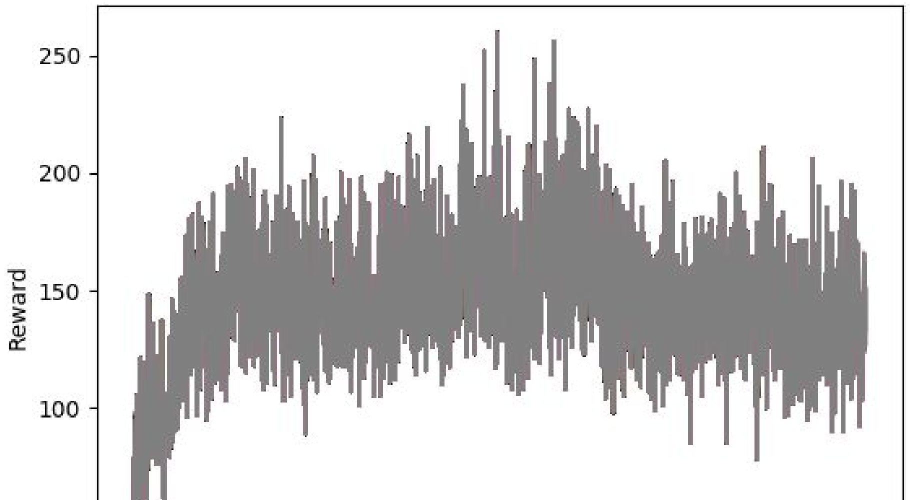
```
Network=(
    self.conv1 = nn.Conv2d(in_channels=4, out_channels=32, kernel_size=8, stride=4)
   self.conv2 = nn.Conv2d(in_channels=32, out_channels=64, kernel_size=4, stride=2)
    self.conv3 = nn.Conv2d(in_channels=64, out_channels=64, kernel_size=3, stride=1)
    self.lin = nn.Linear(in_features=7 * 7 * 64, out_features=512)
    self.actions = nn.Linear(in_features=512, out_features=4)
    self.soft = nn.Softmax(dim=-1)
```

```
        self.value = nn.Linear(in_features=512, out_features=1))
def forward(self, obs):
        h = F.relu(self.conv1(obs))
        h = F.relu(self.conv2(h))
        h = F.relu(self.conv3(h))
        h = h.reshape((-1, 7 * 7 * 64))
        h = F.relu(self.lin(h))
       actions = self.soft(self.actions(h)
        value = self.value(h).reshape(-1)
        return actions, value
```



Analysis : We were able to achieve around 175 score after training for 1500 episodes.

We have tried different input stack sizes of 8 frames. However, the result seems to be improved slightly as shown below.



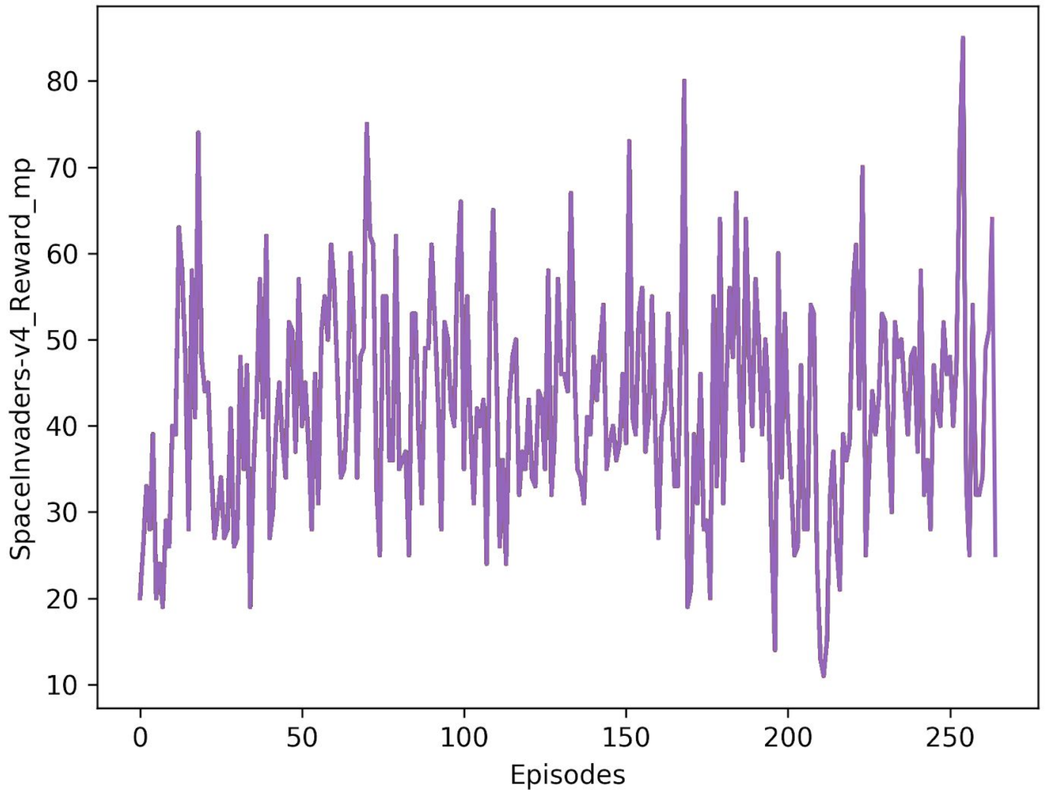
Multi Processing

We have also experimented with using a multi processing in the PPO algorithm. For achieving multi processing ***torch.multiprocessing*** provided by pytorch.

We have used below command

```
args_processes = 20
for rank in range(args_processes):
    p = mp.Process(target=train, args=(sharedpolicy, sharedoptimizer, rank, state_dim, action_dim, n_latent_var, lr, betas, gamma, K_epochs, eps_clip, max_episodes, max_timesteps, log_interval, env_name))
        p.start() ;
for p in processes: p.join()
```

We have also used sharedAdamOptimizer to accumulate gradients for all the workers to update the shared policy network.

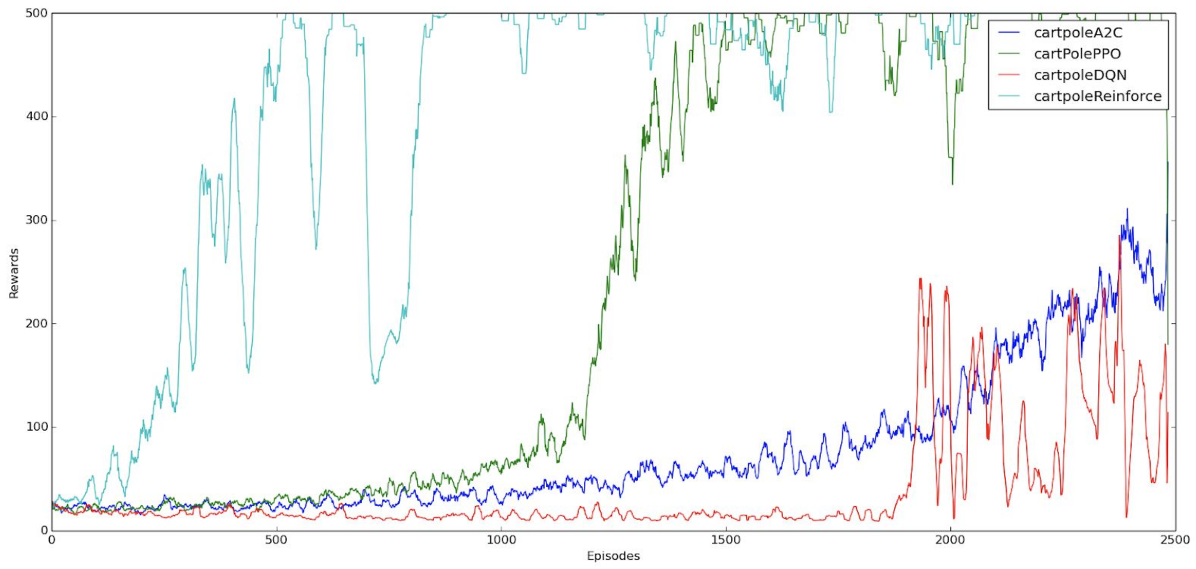


Analysis: We were not able to get it to train correctly. We could only achieve 55 rewards with it. We are not sure what is going wrong.

Training using a GRU:

We also made an attempt to use GRU so that it can model long term dependencies. However, we were unable to get to training properly. Our model was giving bare minimum reward. We are interested to further explore how much GRU can helps in comparison to passing a stack of 4 frames or 8 frames.

2.2 CartPole



2.3

Analysis:

Performance

For a training sequence of 2500 episodes,

- We find that PPO and Reinforce achieve the best performance as shown in the above graph.
- Both PPO and Reinforce gave a consistent reward of 500 after some iterations.

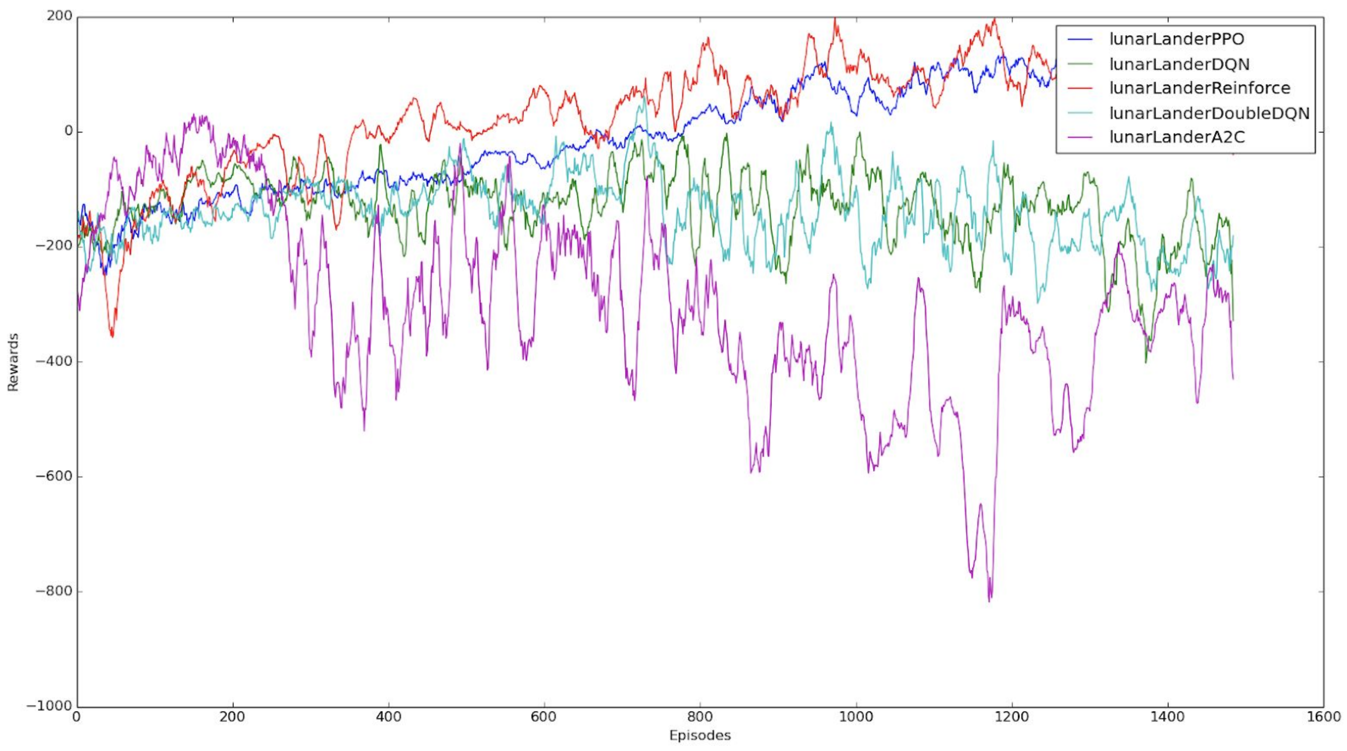
Efficiency (time taken)

- From our analysis on the amount of time taken among the algorithms to achieve the solved reward, we find that Reinforce achieves it around 900 episodes, while the second best is PPO which achieves it in 1500 episodes.

From the graph, we observe that PPO though has less reward till 1000 episodes has faster increase in reward later and achieves best reward in about 1500 episodes.

More importantly, PPO is way faster than the other algorithms. It converges in just 250secs.

2.4 LunarLander



Analysis:

Performance

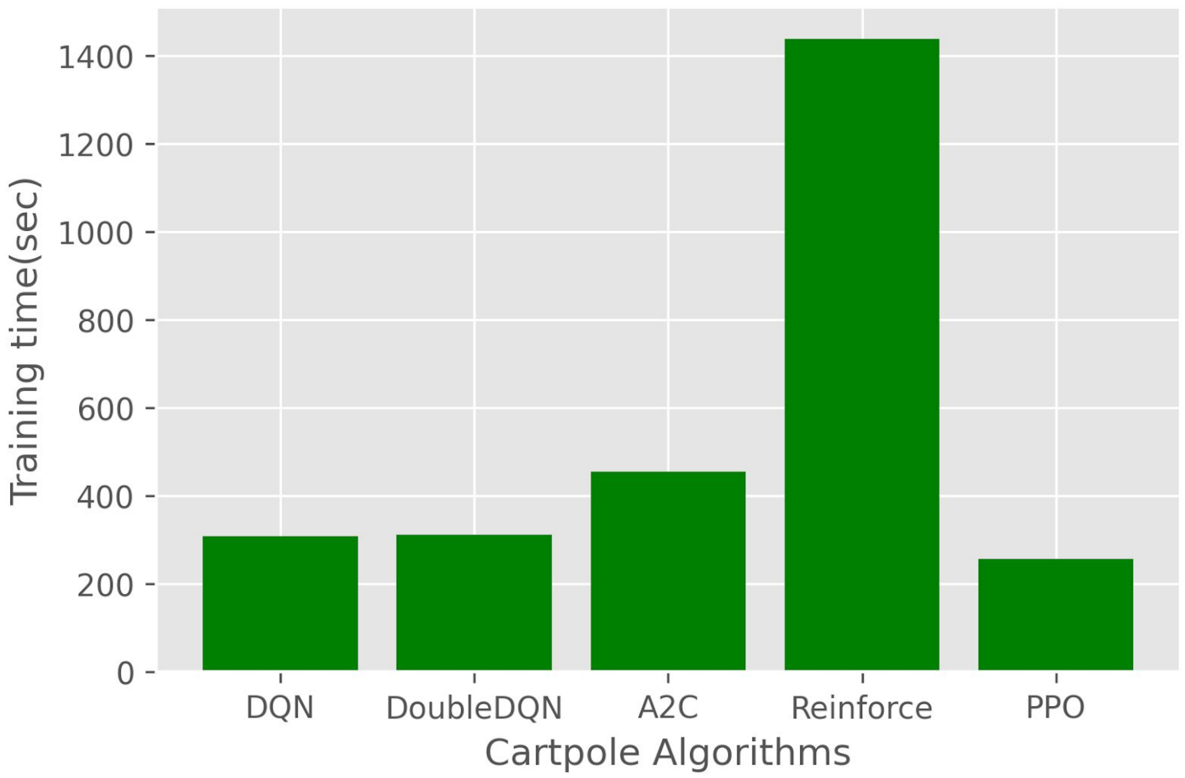
For a training sequence of 1500 episodes,

- We find that PPO and Reinforce achieve the best performance as shown in the above graph.
- This is consistent with our performance results during our cartpole environment result
- Both PPO and Reinforce gave a consistent reward of 200 after some iterations.

Efficiency (time taken)

- From our analysis on the amount of time taken among the algorithms to achieve the solved reward, we find that both Reinforce and PPO converge to a reward of 200 within 1500 episodes, while other models seem to be saturated as their reward accumulation over episodes curve has flattened.
- However, PPO takes comparatively less time to converge.

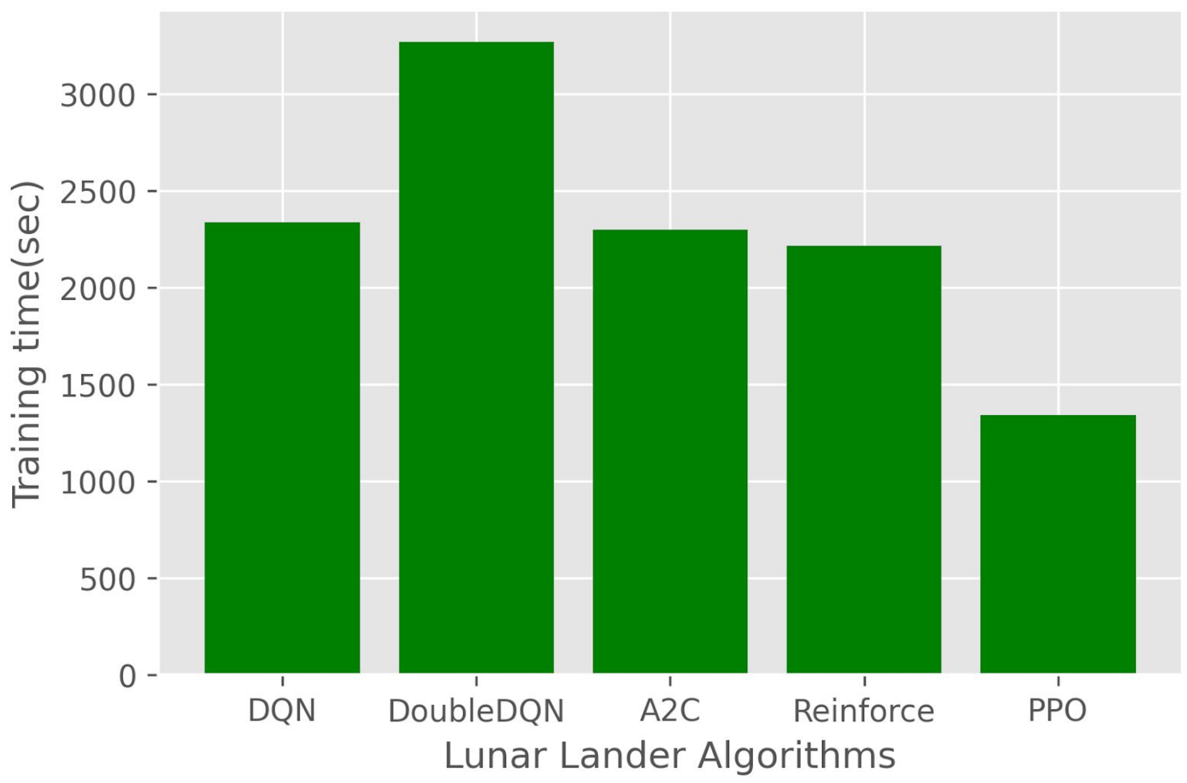
Analysis on Training time across algorithms



Analysis

As shown in the above figure, we do an analysis on the training time taken by various algorithms for CartPole environment for training over 2500 episodes.

- It can be noted that Reinforce and PPO which have given best performance in successfully solving the CartPole task take the maximum training time.
- PPO and Reinforce take double the training time taken by DQN, DoubleDQN and A2C algorithm.



Analysis

As shown in the above figure, we do an analysis on the training time taken by various algorithms for CartPole environment for training over 1600 episodes.

- It can be noted that PPO which have given best performance in successfully solving the CartPole task take the minimum training time.
- Reinforce while taking similar time to DQN and A2C achieves better performance much faster.

Conclusion

Our analysis for cartpole and LunarLander environment shows that PPO is best algorithm that gives very good performance without taking much time. Our further analysis on image based SpaceInvaders shows decent results using PPO.