Comparing deep reinforcement algorithm on lunar space lander.

Sai Charith Reddy Pasula
50320452
saichari@buffalo.edu
Department of Computer Science
University at buffalo

Nikhil Tej Kantu
50336812
nikhilte@buffalo.edu
Department of Robotics
University at buffalo

Abstract

The aim of the assignment is to compare the different deep reinforcement algorithm on the lunar lander environment and Acrobot environment provided by openAI gym.

1 Introduction

1.1 DQN

we approximate the Q- values by adding a term w to it.

$$Q(s, a, w) \approx Q^{\pi}(s, a)$$

Q value approximation

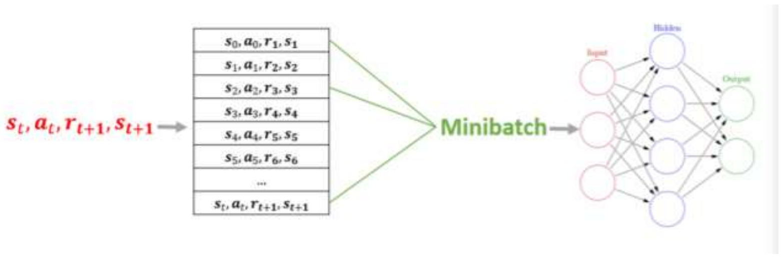
We minimize w by using the objective function:

$$\mathcal{L}(w) = \mathbb{E}\left[\left(\underbrace{r + \gamma \max_{a'} Q(s', a', w)}_{\text{target}} - Q(s, a, w)\right)^2\right]$$

Objective function

We optimise this objective function by using stochastic gradient descent.
Deepmind team gave three different optimizations to the original Deep RL techniques:

- The data in the RL environment is sequential in nature. This is because the agent performs one action and the next action is related to the state he ends up in. So, the input to the network is correlated. To overcome this problem, we use experience replay. In experience replay, the agent performs an action according to the epsilon-greedy policy. We then store the state, action, reward, next_state tuple in replay memory. we sample a minibatch of transitions from the memory to train the network. This helps in breaking the correlation between the inputs. I used the python Deque collection to implement this.

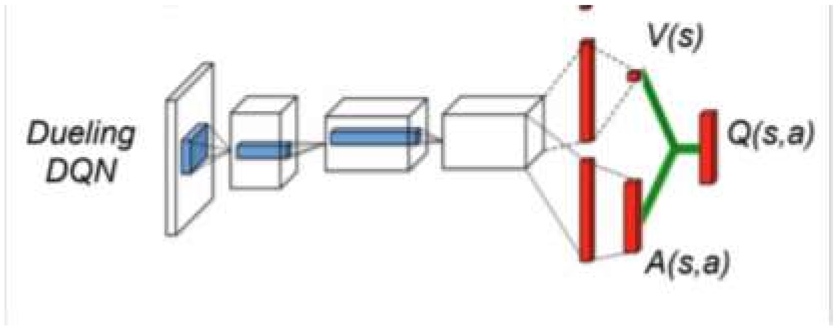


Experience Replay

- Policy changes rapidly with slight changes to Q-values, due to this, the distribution of data can swing from one extreme to another. To overcome this, we use two kinds of networks, the original network (w) and the target network (w^-). We use the target model to predict the Q-values. After C updates, we synchronize w^- to w. By doing this, we fix the Q-value targets temporarily to avoid a moving target.
- Scale of rewards and Q-values is unknown. This we have large rewards, the Q-values increase a lot due to this to prevent this, we clip the rewards between -1 and 1. To do this, I used np.sign function.

1.2 Dueling DQN:

Q- Values tells us how good it is to take an action a in a state s. We can decompose Q(s,a) as A(s,a) +V(s). Where A(s,a) is called Advantage function. If A>0 our gradient is pushed in that direction. Else gradient is pushed in the opposite direction. In dueling DQN, we separate the estimator using two new streams, one estimates the state value V(S) and one estimates the advantage for each action A(s,a).



Dueling DQN

While combining the two streams, we can directly add the two outputs:

$$Q(s,a;\theta,\alpha,\beta) = V(s;\theta,\beta) + A(s,a;\theta,\alpha)$$

But from this if we are given a Q, we cannot recover V and A uniquely which in turn leads to poor practical performance.

Solutions:

$$Q(s,a;\theta,\alpha,\beta) = V(s;\theta,\beta) + \left(A(s,a;\theta,\alpha) - \max_{a' \in |A|} A(s,a';\theta,\alpha)\right)$$

This forces the advantage function to have zero advantage at chosen action.

We can also replace the max with the average:

$$Q(s,a;\theta,\alpha,\beta) = V(s;\theta,\beta) + \left(A(s,a;\theta,\alpha) - \frac{1}{|A|}\sum_{a'} A(s,a';\theta,\alpha)\right)$$

This increases the stability of the optimizations. I implemented the average of the Advantage function in this code.

1.3 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 directly approximate the policy directly.

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

Policy approximation

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

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

Gradient Ascent

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

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

Sigmoid activation to get probability of actions

The finally gradient of J is written as

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

Gradient of J

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

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

Reinforce update function

Keras does not have a built-in loss function to handle log-likelihood loss. So, we define a custom loss function that calculates the log likelihood of the predicted values and the true values and multiplies them with the advantages and sums them up.

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.

1.4 Actor-Critic

Reinforce has high variance to overcome this, we use a critic to estimate the action value function along with the policy approximation. We can think of this as the actor is the agent who does whatever action he wants and critic is tells the actor if that action was good or not.

The actor updates theta and the critic updates w. The Q-values that the critic predicts is sent to the actor using which the actor knows if this was a good action or not. The actor uses a log likelihood loss similar to policy gradient and the actor uses mean squared error as loss function.

The actor model takes state and delta(the TD error calculated using critic) and the critic takes just the state as input and outputs the Q value for the state action pair.

The choose action function is similar to reinforce it gets the probabilities from the actor and chooses a random action based on the probabilities.

The learn function calculates the critic values for the preset state and next state and calculates the target and the td error. The actor is trained using this td error and the critic is trained using the state and target.

1.5 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.

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

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 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 0.2 in the PPO paper. This is very simple to implement.

1.6 Environments:

In this project, I have trained the agent in one environments for comparing the policy rewards:

- **LunarLander-v2:**

Landing pad is always at coordinates (0, 0). Coordinates are the first two numbers in state vector. Reward for moving from the top of the screen to landing pad and zero speed is about 100...140 points. If lander moves away from landing pad it loses reward back. Episode finishes if the lander crashes or comes to rest, receiving additional -100 or +100 points. Each leg ground contact is +10. Firing main engine is -0.3 points each frame. Solved is 200 points. Landing outside landing pad is possible. Fuel is infinite, so an agent can learn to fly and then land on its first attempt. Four discrete actions available: do 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 right orientation engine]

- **Acrobot-v1:**

Acrobot is a 2-link pendulum with only the second joint actuated. Initially, both links point downwards. The goal is to swing the end-effector at a height at least the length of one link above the base. Both links can swing freely and can pass by each other, i.e., they don't collide when they have the same angle.

State:

The state consists of the sin () and cos () of the two rotational joint angles and the joint angular velocities:
[cos (theta1) sin (theta1) cos (theta2) sin (theta2) thetaDot1 thetaDot2]. For the first link, an angle of 0 corresponds to the link pointing downwards. The angle of the second link is relative to the angle of the first link. An angle of 0 corresponds to having the same angle between the two links. A state of [1, 0, 1, 0…] means that both links point downwards.

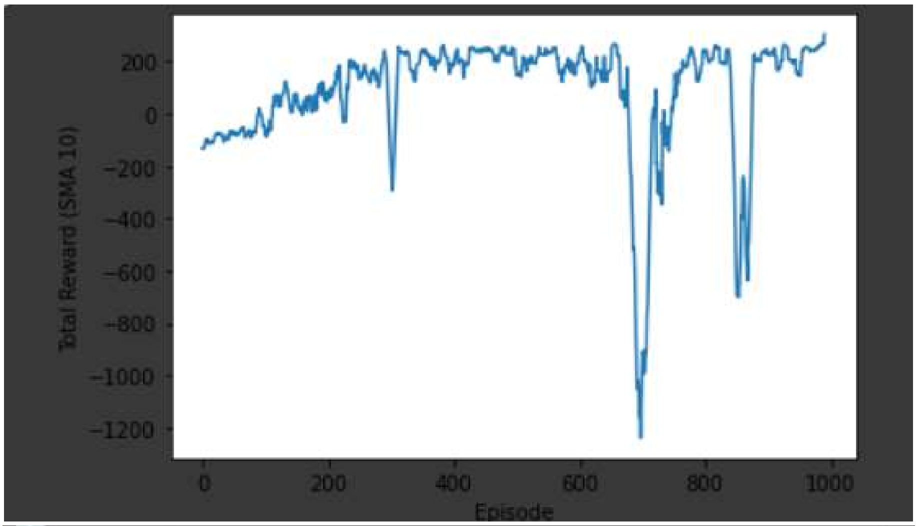
Actions:

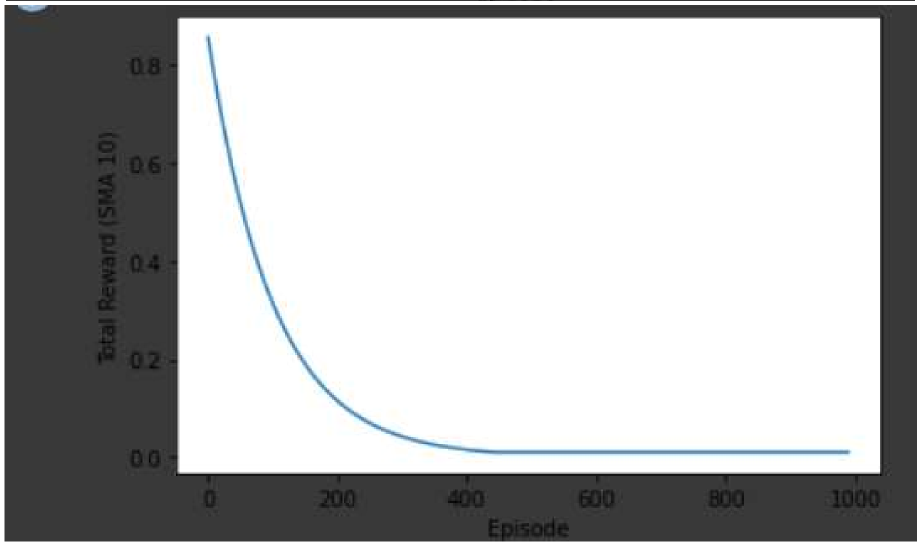
The action is either applying +1, 0 or -1 torque on the joint between the two pendulum links.

2 Results:

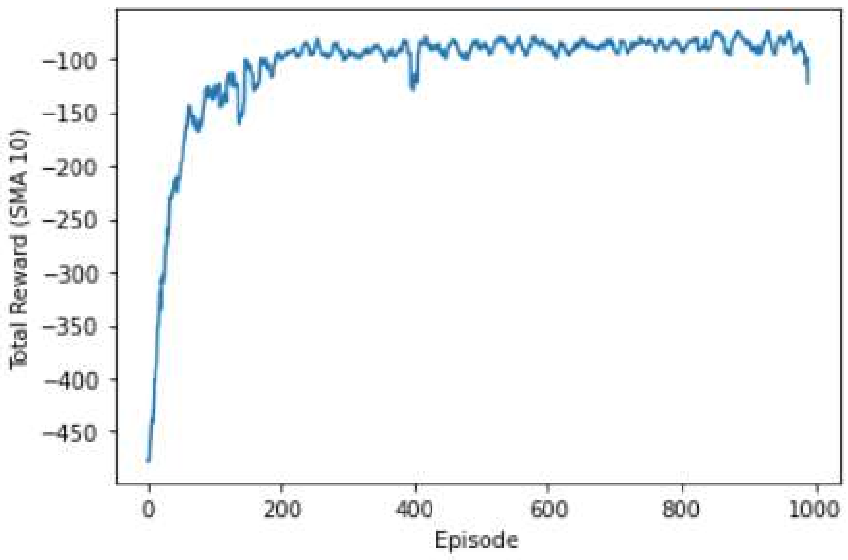
2.1 DQN:

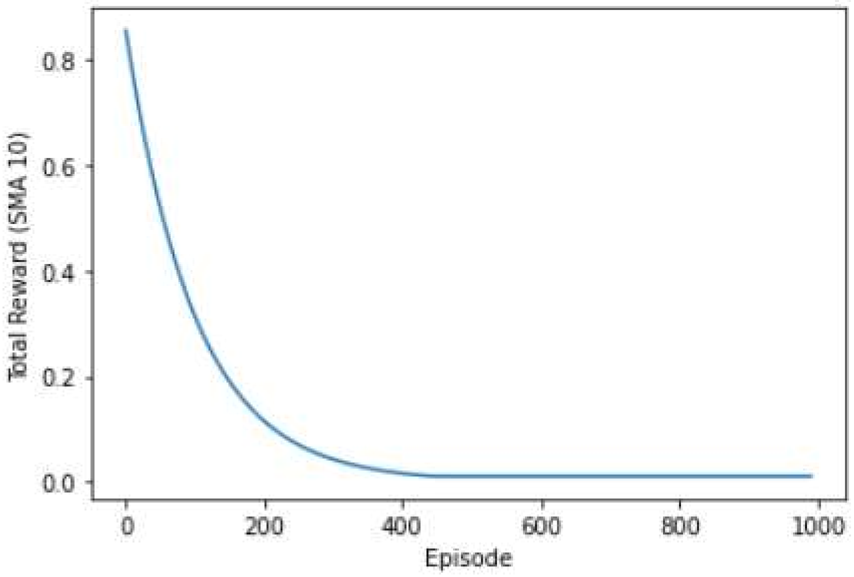
- **LunarLander-v2**





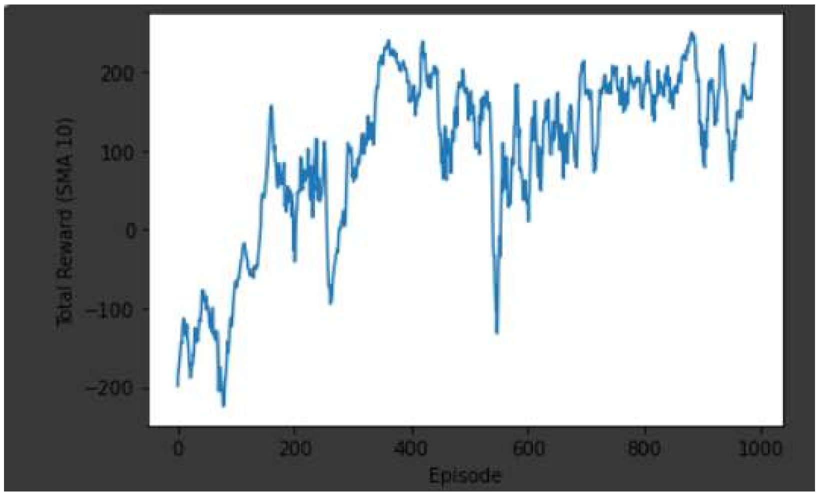
- **Acrobot-v1**



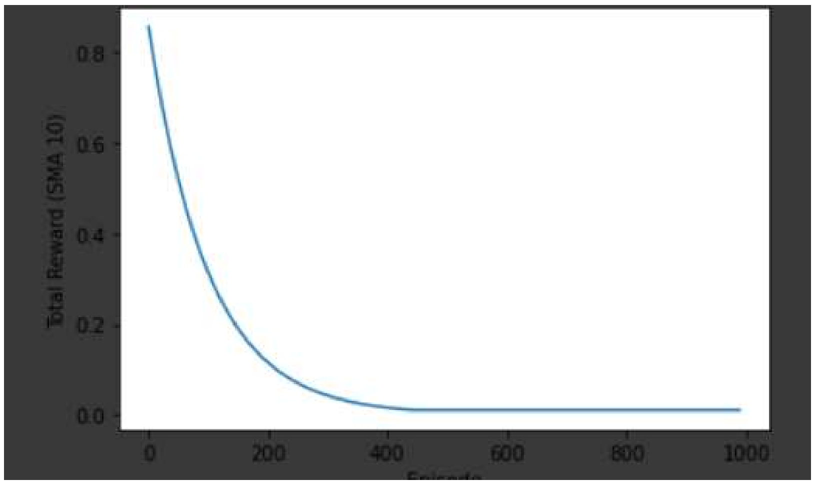


2.2 Dueling DQN:

- **LunarLander-v2**

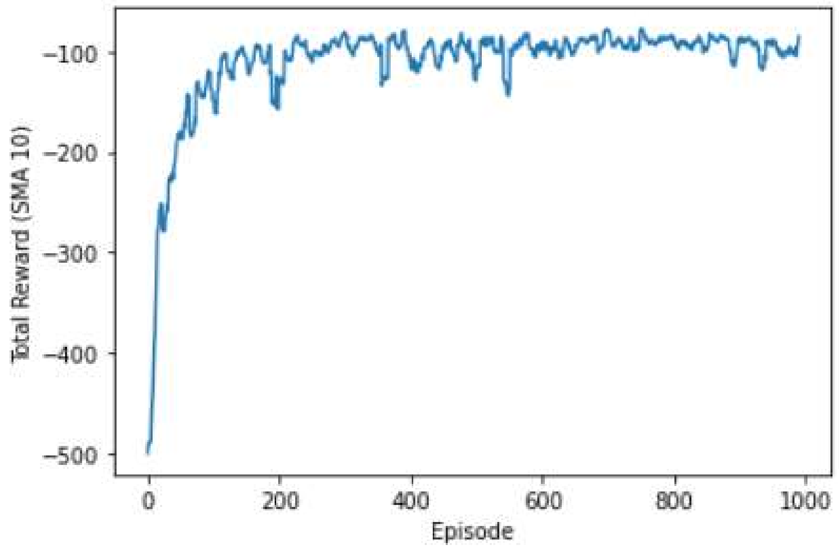

Total Reward (SMA 10)
Episode
200
100
0
−100
−200
0
200
400
600
800
1000

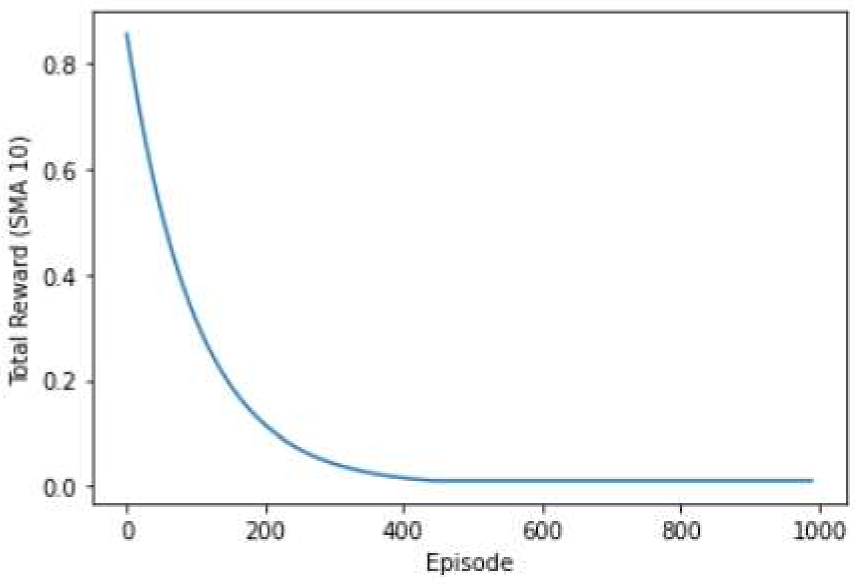
•


Total Reward (SMA 10)
0.8
0.6
0.4
0.2
0.0
0
200
400
600
800
1000

•

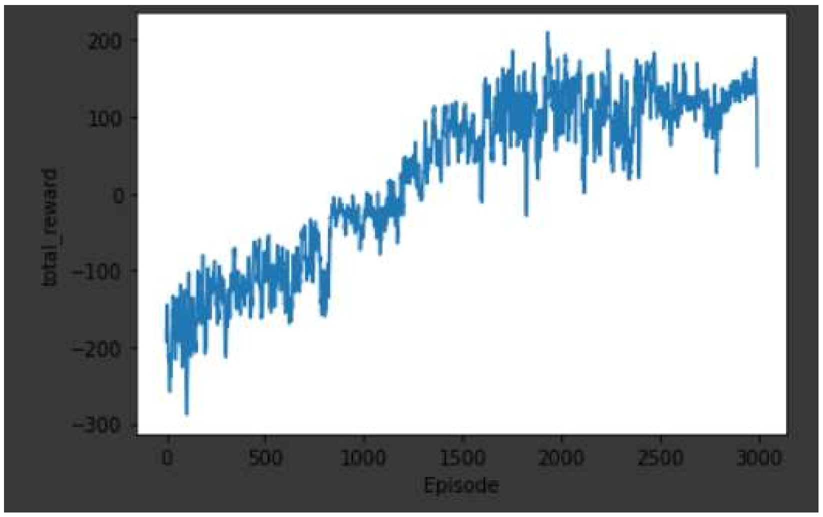
- **Acrobot-v1**


Total Reward (SMA 10)
Episode
−100
−200
−300
−400
−500
0
200
400
600
800
1000

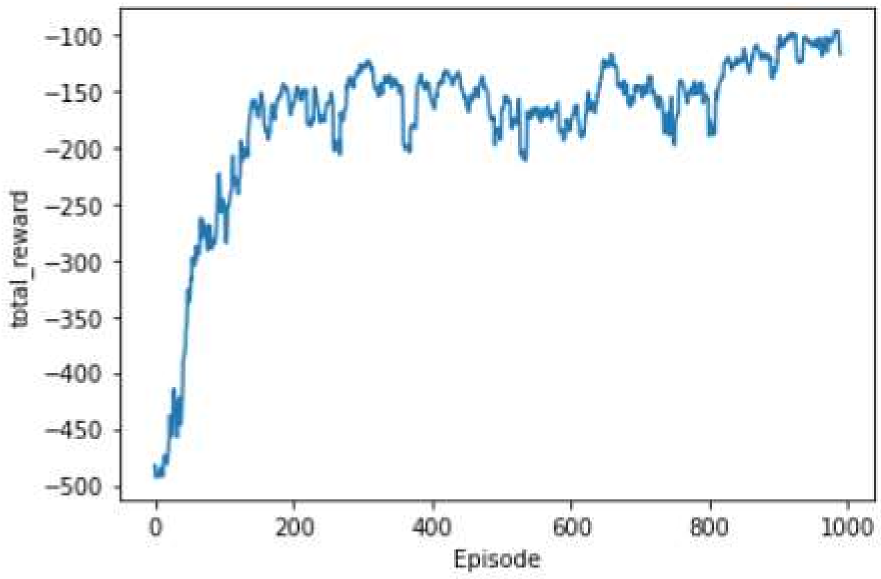


2.3 Reinforce:

- **LunarLander-v2**

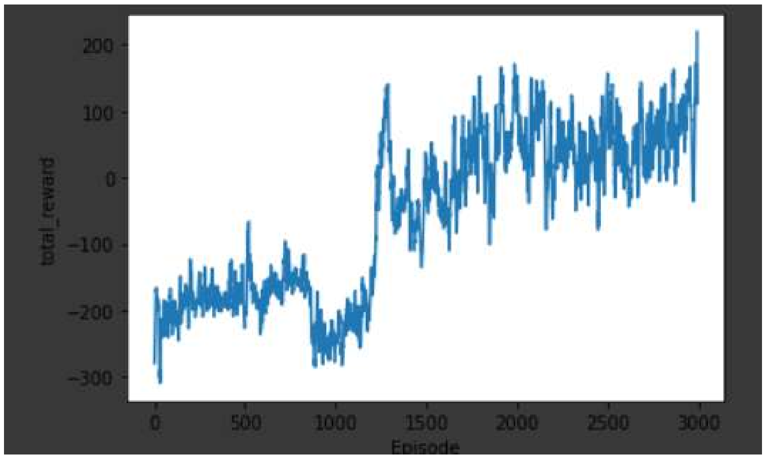


- **Acrobot-v1**

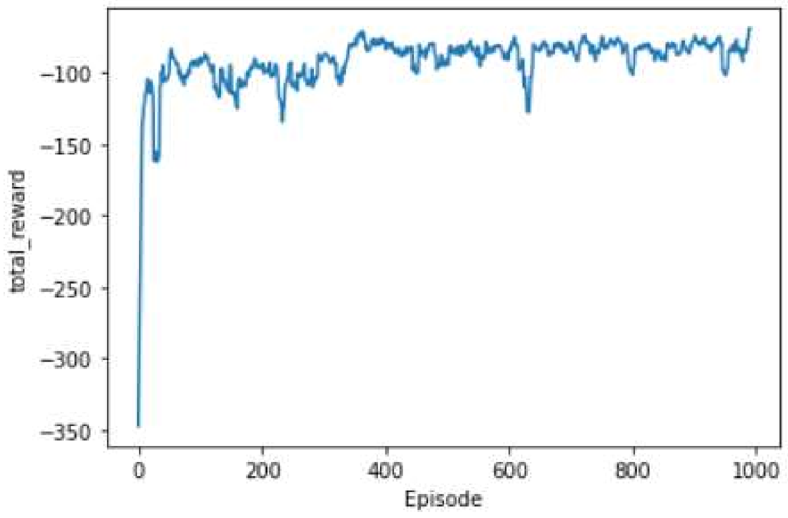


2.4 Actor-Critic:

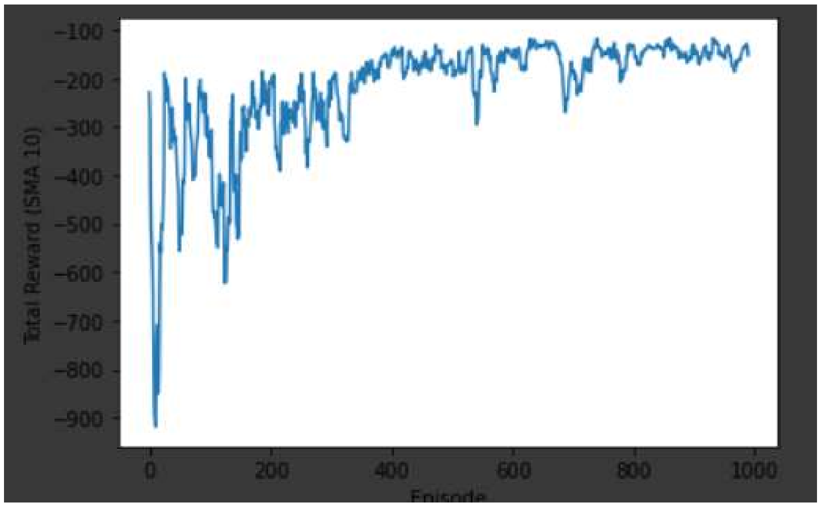
- **LunarLander-v2**



- **Acrobot-v1**



2.5 PPO:



3 Conclusion:

From the above graphs we can infer the following thing:

- DQN trains very fast somewhere around 200 episodes but it isn't as stable as dueling DQN. DQN overestimates a few actions.
- DQN oscillates to negative reward suddenly and then comes back the max reward whereas dueling DQN changes a lot but only in the positive reward region.
- Policy gradients algorithms like REINFORCE and 1 step AC need a lot of samples to converge.
- REINFORCE has a lot of variance as seen from the graph and AC slightly decreases the variance.
- PPO is by far the most stable Actor-Critic algorithm but it needs lot of samples to converge.
- All of the policy gradient algorithms and AC critic algorithms are highly

sensitive to hyper parameter tuning. I am sure we can get better results for some different set of hyper parameters.

4 References

1) https://gym.openai.com/envs/LunarLander-v2/

2) Lecture slides – CSE510 Introduction to Reinforcement Learning

3)https://medium.com/@jonathan_hui/rl-proximal-policy-optimization-ppo-explained-77f014ec3f12

4) Human-level control through deep reinforcement learning (Mnih et. al)

5) Dueling Network Architectures for Deep Reinforcement Learning (Wang et.al)

6)https://towardsdatascience.com/understanding-actor-critic-methods-931b97b6df3f?gi=699deab97419

7) Proximal Policy Optimization Algorithms https://arxiv.org/pdf/1707.06347.pdf