Programming Assignment - 2

Kamalesh Kumar ce20b054
Kamalesh M ce20b055

March 2023

This report consists of the discussions and experimentations conducted on DQN and Actor-Critic frameworks implemented in the following environments:

- Acrobot
- CartPole
- MountainCar

1 DQN

The metric for performance is considered as the no of episodes taken to achieve a threshold reward which is pre-defined for each environment. The initial configuration of hyper-parameters from which we need to improve is taken from tutorial-4. The hyper-parameters are changed till we get 5 improving configurations.

1.1 Acrobot

For Acrobot environment, the performance is set as the no of episodes taken to reach an average reward of -100.

Initial configuration of hyper-parameters as given in tutorial-4 required 548 episodes to attain a average reward of -100.

- Default configuration of hyper-parameters :
  - Buffer size = 10^5
  - Batch size = 64
  - $\gamma = 0.99$
  - Learning rate = 5×10^{-4}
  - Update Every = 20 (Frequency of update for the target network)
  - NN : Two hidden layers with 128 and 64 neurons.
  - No.of episodes to solve: 548

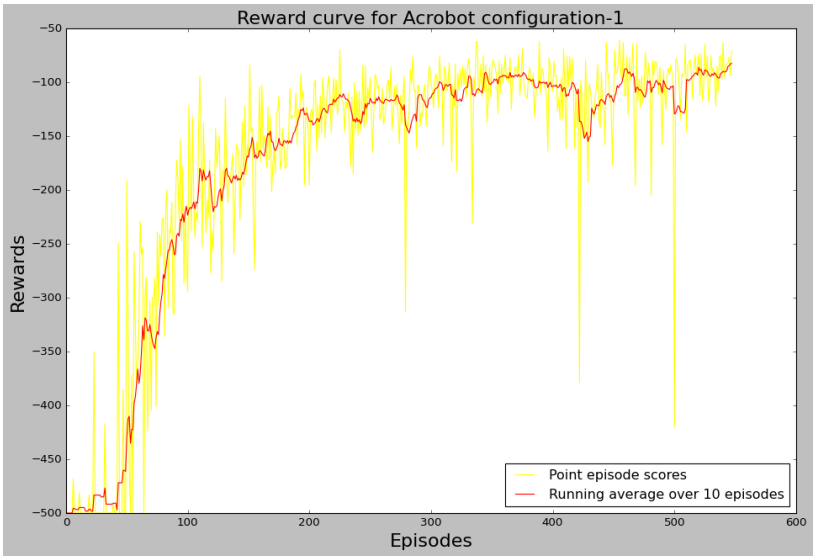

Figure 1: Reward curve for DQN in Acrobot– configuration 1

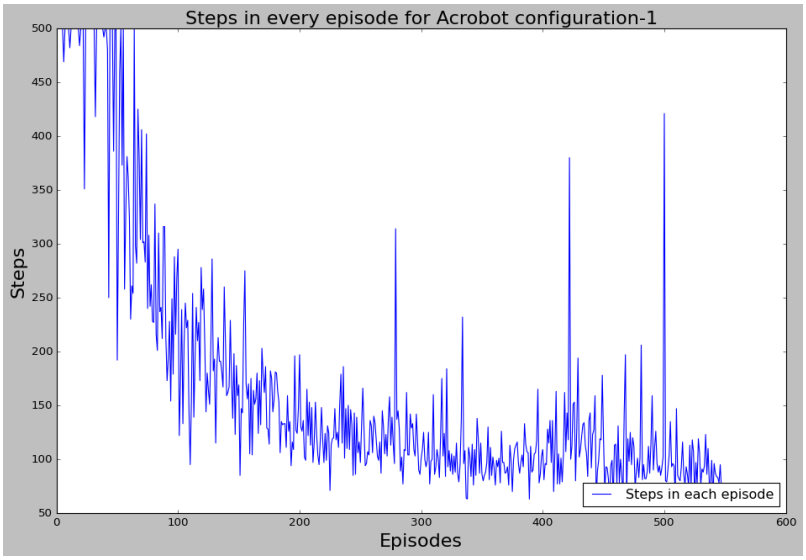

Figure 2: Plot of no.of steps per episode for DQN in Acrobot– configuration 1

- Configuration-2 of hyper-parameters : Learning rate is increased from 5×10^{-4} to 10^{-3}. Clearly, increasing the learning rate by a small margin boosts performance, which is also trivial.
  - Buffer size = 10^5

- Batch size = 64
- $\gamma = 0.99$
- Learning rate = 10^{-3}
- Update Every = 20 (Frequency of update for the target network)
- NN : Two hidden layers with 128 and 64 neurons.
- No.of episodes to solve: 443

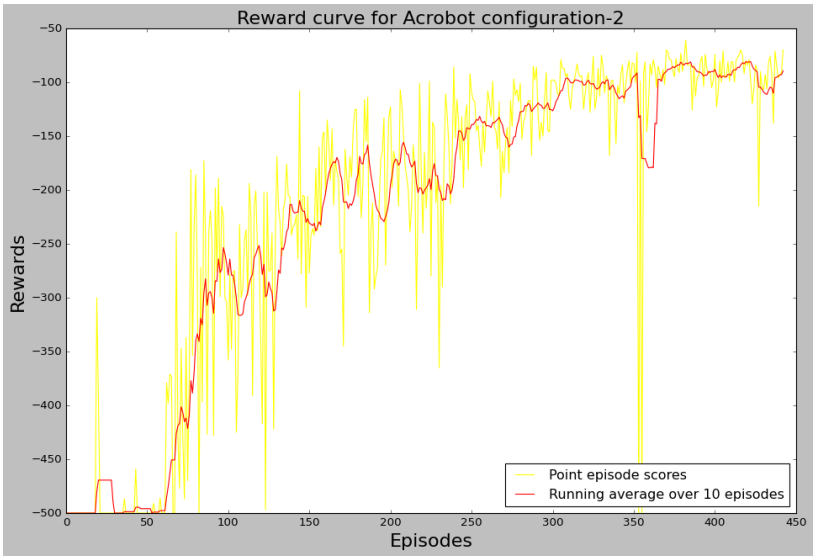

Figure 3: Reward curve for DQN in Acrobot– configuration 2

- Configuration-3 of hyper-parameters :

  Decreasing the NN's complexity and buffer size from 10^5 to 10^4 gave a slightly better performance.

  - Buffer size = 10^4
  - Batch size = 64
  - $\gamma = 0.99$
  - learning rate = 10^{-3}
  - Update Every = 20
  - NN : Two hidden layers with 64 and 64 neurons.
  - No.of episodes to solve: 389

- Configuration-4 of hyper-parameters : Buffer size is increased from 10^4 to 10^5

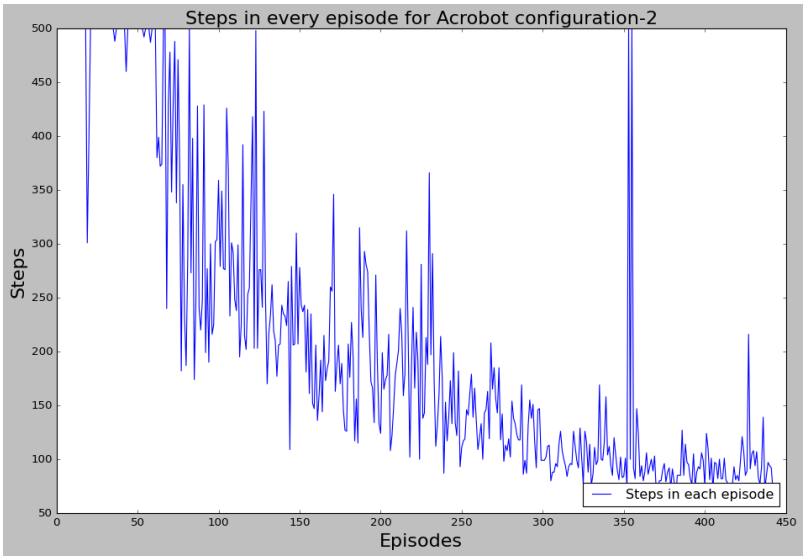

Figure 4: Plot of no.of steps per episode for DQN in Acrobot– configuration 2

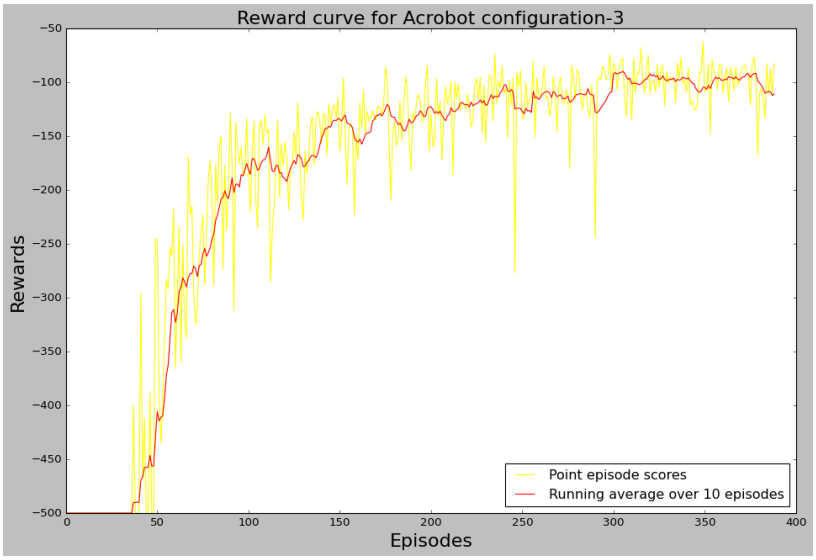

Figure 5: Reward curve for DQN in Acrobot– configuration 3

- Buffer size = 10^5
- Batch size = 64
- $\gamma = 0.99$
- learning rate = 10^{-3}

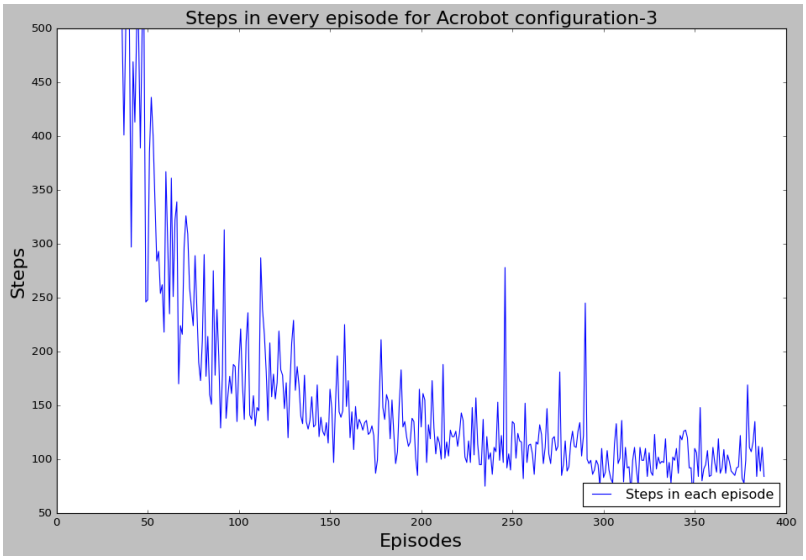

Figure 6: Plot of no.of steps per episode for DQN in Acrobot– configuration 3

- Update Every = 20
- NN : Two hidden layers with 64 and 64 neurons.
- No.of episodes to solve: 383

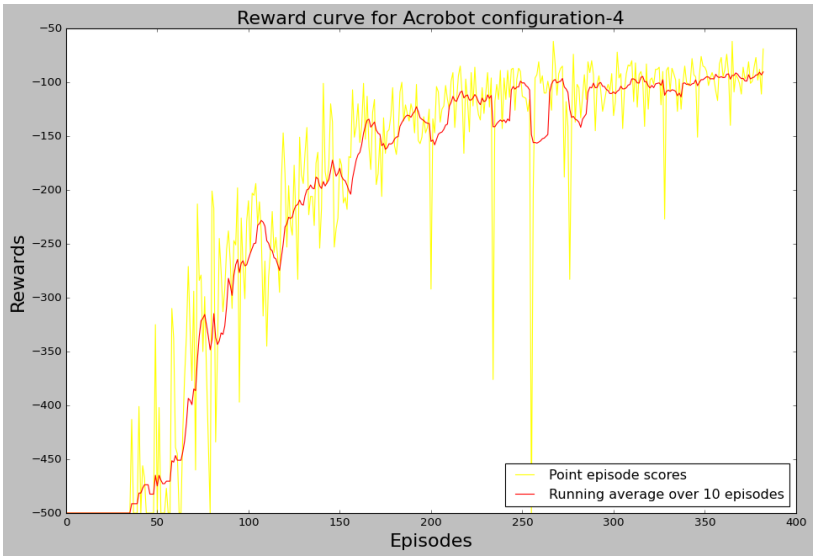

Figure 7: Reward curve for DQN in Acrobot– configuration 4

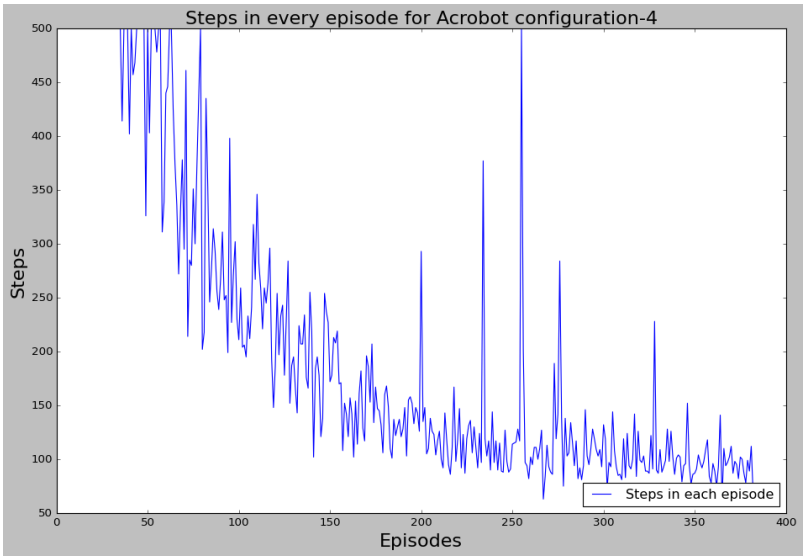

Figure 8: Plot of no.of steps per episode for DQN in Acrobot– configuration 4

- Configuration-5 of hyper-parameters :

  Batch size is increased from 64 to 128 reduced the number of episodes required to solve, which could be due to the reducing variance with increasing batch size. Ideally, batch size equal to the buffer size will have the lowest variance.

  - Buffer size = 10^5
  - Batch size = 128
  - $\gamma = 0.99$
  - learning rate = 10^{-3}
  - Update Every = 20
  - NN : Two hidden layers with 64 and 64 neurons.
  - No.of episodes to solve: 360

1.2 CartPole

For CartPole environment, the performance is set as the no of episodes taken to reach an average reward of 200.

Initial configuration of hyper-parameters as given in tutorial-4 required 1012 episodes to attain a average reward of 200.

- Default configuration of hyper-parameters :

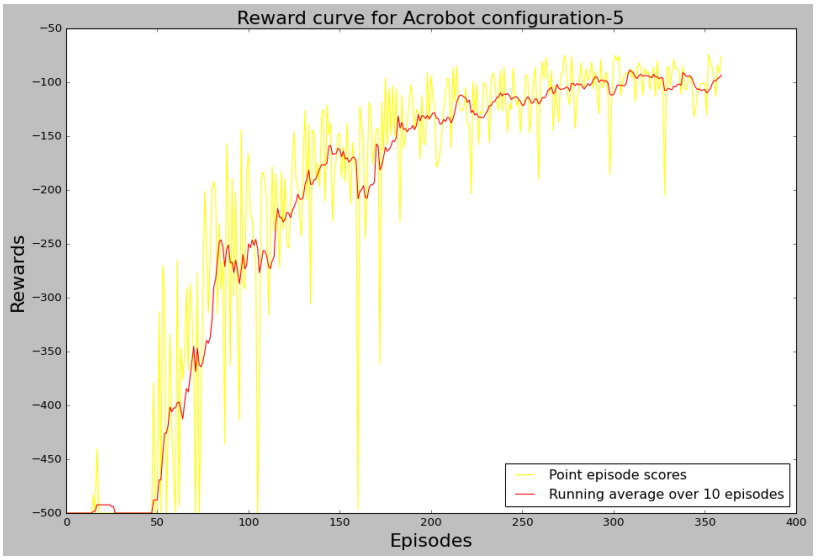

Figure 9: Reward curve for DQN in Acrobot– configuration 5

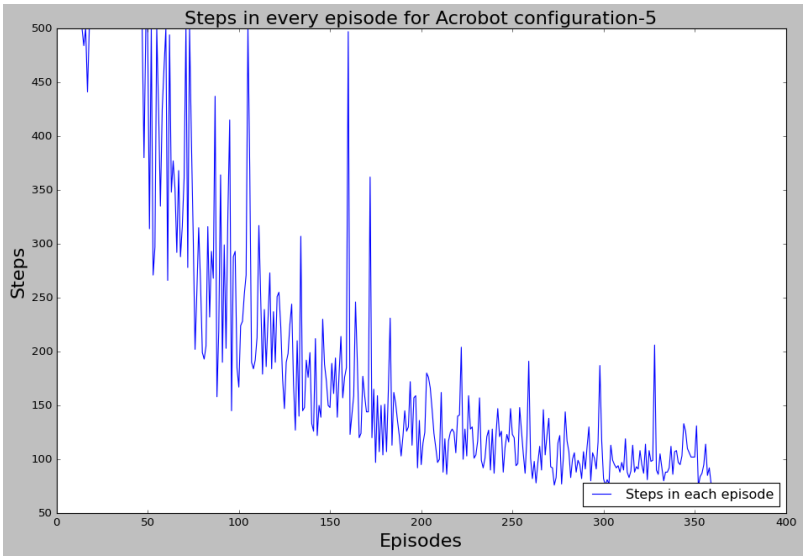

Figure 10: Plot of no.of steps per episode for DQN in Acrobot– configuration 5

- Buffer size = 1e5
- Batch size = 64
- $\gamma = 0.99$
- learning rate = 5e-4

– Update Every = 20
– NN : Two hidden layers with size 128 and 64.
– No.of episodes to solve: 1012

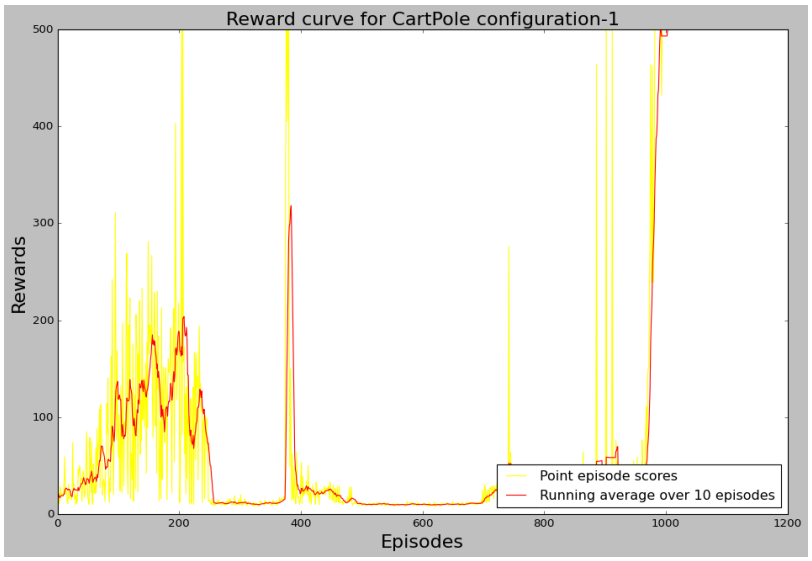

Figure 11: Reward curve for DQN in Cartpole– configuration 1

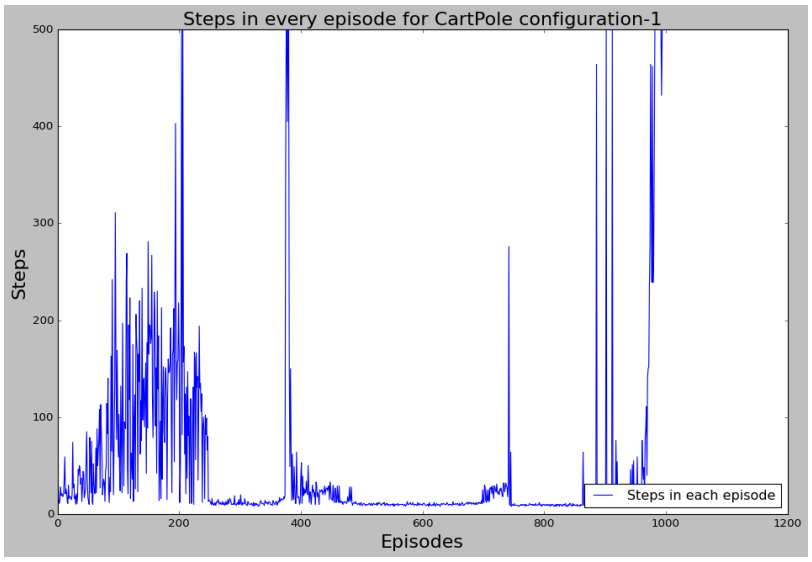

Figure 12: Plot of no.of steps per episode for DQN in Cartpole– configuration 1

- Configuration-2 of hyper-parameters : Increasing the batch size from 64 to 128 reduced the number of episodes required to solve, which could be due to the reducing variance with increasing batch size. Ideally, batch size equal to the buffer size will have the lowest variance.
  - Buffer size = 1e5
  - Batch size = 128
  - $\gamma = 0.99$
  - learning rate = 5e-4
  - Update Every = 20
  - NN : Two hidden layers with size 128 and 64.
  - No.of episodes to solve: 639

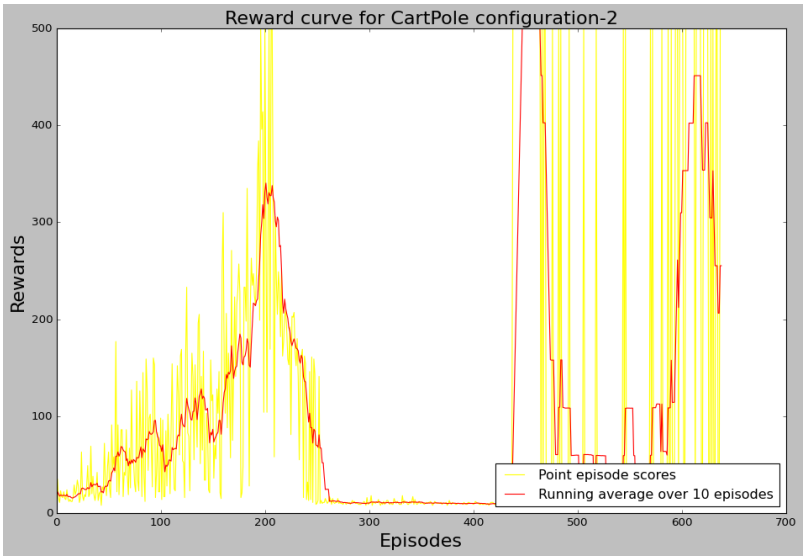

Figure 13: Reward curve for DQN in Cartpole– configuration 2

- Configuration-3 of hyper-parameters : Increasing the size of NN from 128,64 to 256,128 improved the performance compared to the previous configuration.
  - Buffer size = 1e5
  - Batch size = 128
  - $\gamma = 0.99$
  - learning rate = 5e-4
  - Update Every = 20

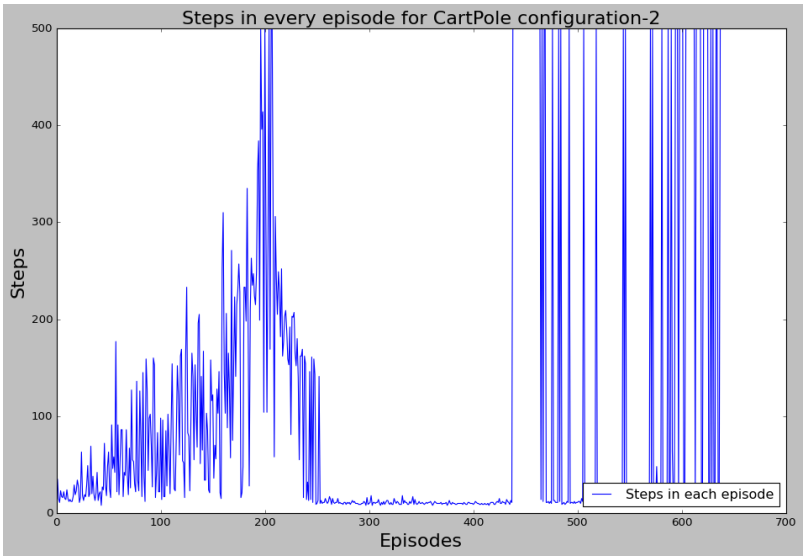

Figure 14: Plot of no.of steps per episode for DQN in Cartpole– configuration 2

- NN : Two hidden layers with size 256 and 128.
- No.of episodes to solve: 303

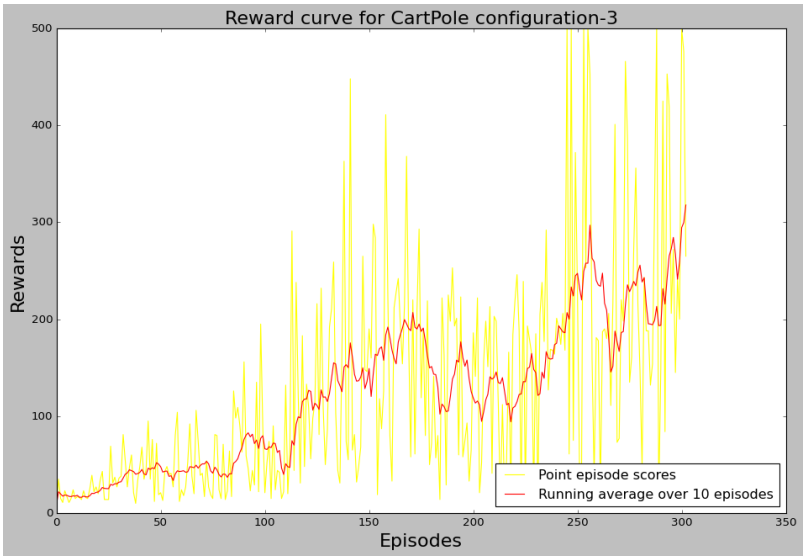

Figure 15: RReward curve for DQN in Cartpole– configuration 3

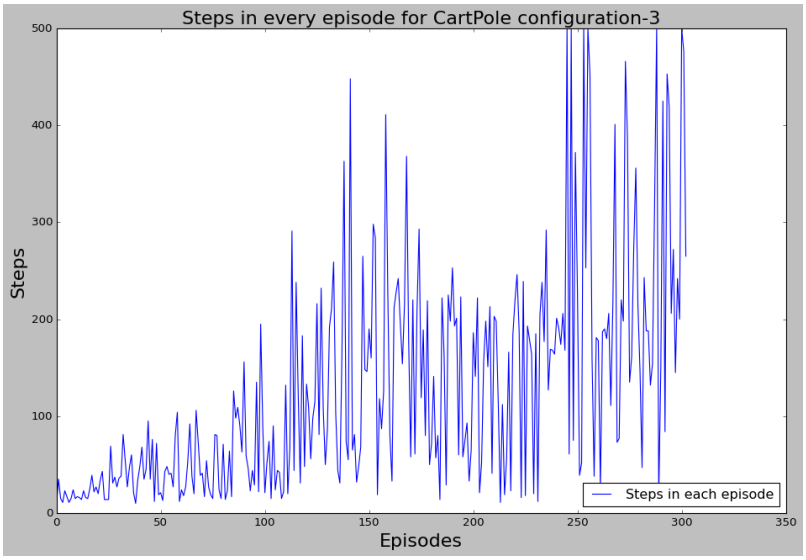

Figure 16: Plot of no.of steps per episode for DQN in Cartpole– configuration 3

- Configuration-4 of hyper-parameters : Increasing the size of NN further from 256,128 to 256,256 gave even better results.
  - Buffer size = 1e5
  - Batch size = 128
  - $\gamma = 0.99$
  - learning rate = 5e-4
  - Update Every = 20
  - NN : Two hidden layers with size 256 and 256.
  - No.of episodes to solve: 284

- Configuration-5 of hyper-parameters : Decreasing the buffer size from 10^5 to 10^4.
  - Buffer size = 1e4
  - Batch size = 128
  - $\gamma = 0.99$
  - learning rate = 5e-4
  - Update Every = 20
  - NN : Two hidden layers with size 256 and 256.
  - No.of episodes to solve: 242

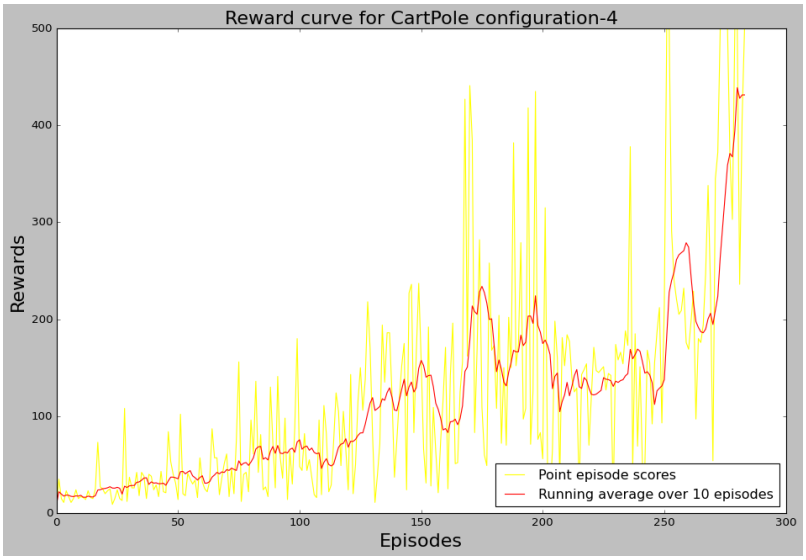

Figure 17: Reward curve for DQN in Cartpole– configuration 4

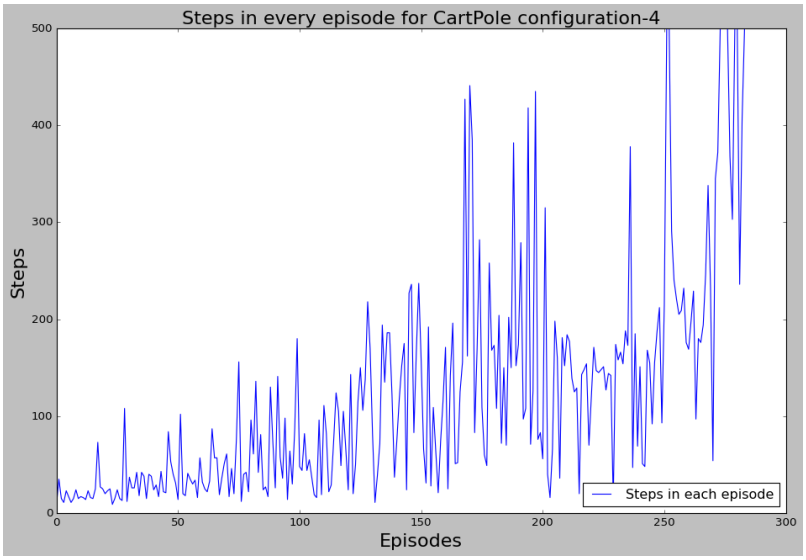

Figure 18: Plot of no.of steps per episode for DQN in Cartpole– configuration 4

1.3 Mountaincar

Initial configuration of hyper-parameters as given in tutorial-4 didn't attain a average reward of -150 with a considerable performance.For MountainCar environment, the performance is set as the no of episodes taken to reach an

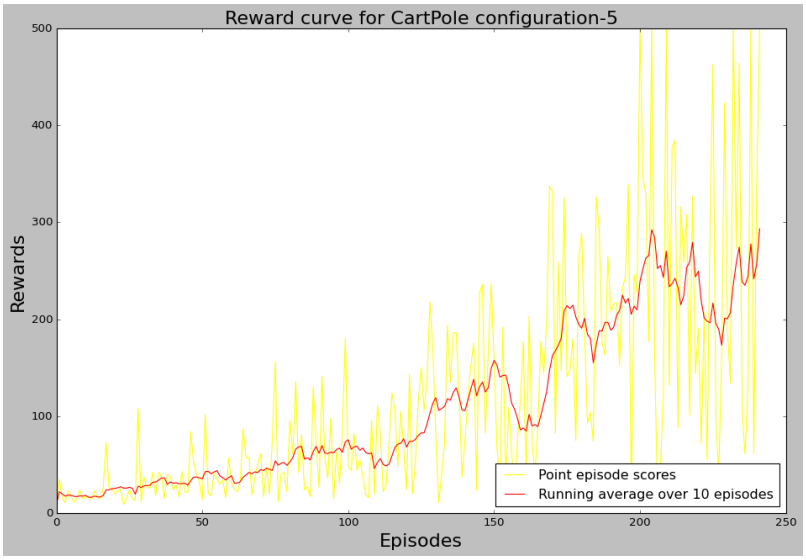

Figure 19: Reward curve for DQN in Cartpole– configuration 5

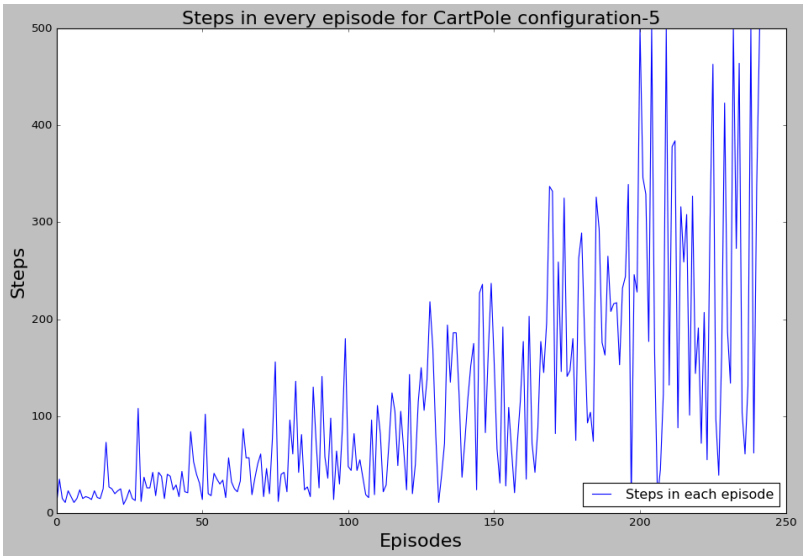

Figure 20: Plot of no.of steps per episode for DQN in Cartpole– configuration 5

average reward of -150. For more than 1000 episodes, the agent was unsuccessful in the solving the environment in many configurations.

- Default configuration of hyper-parameters :
  - Buffer size = 1e5

  - Batch size = 64
  - $\gamma = 0.99$
  - learning rate = 5e-4
  - Update Every = 20
  - NN : Two hidden layers with size 128 and 64.
  - No.of episodes to solve: Did not solve

- Configuration-2 of hyper-parameters : Increasing the learning from 5x10^-4 to 10^-1. Making this change slightly improved the performance, compared to the previous configuration, which was never able to reach the goal state.
  - Buffer size = 1e5
  - Batch size = 64
  - $\gamma = 0.99$
  - learning rate = 1e-1
  - Update Every = 20
  - NN : Two hidden layers with size 128 and 64.
  - No.of episodes to solve: Did not solve

- Configuration-3 of hyper-parameters : Increasing the buffer size from 10^5 to 10^6 and decreasing the learning rate from 10^{-1} to 10^{-2}. Increasing the buffer size by 10 times encouraged the agent to learn from trajectories far into the past, decreasing the learning rate by ten times finally solved this environment!
  - Buffer size = 1e6
  - Batch size = 64
  - $\gamma = 0.99$
  - learning rate = 1e-2
  - Update Every = 20
  - NN : Two hidden layers with size 128 and 64.
  - No.of episodes to solve: 713

Many other configurations were tested for the MoutainCar environment, but they were not successful in solving the environment.

2 Code snippet for DQN

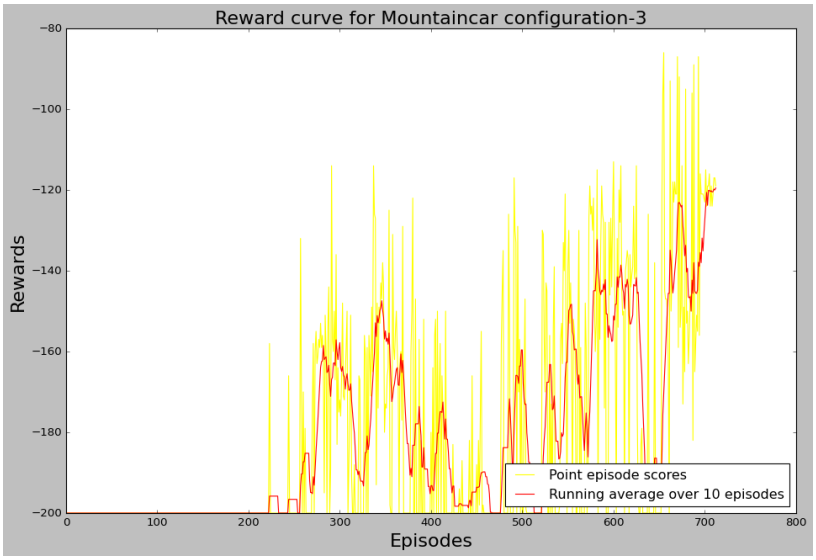

Figure 21: Reward curve for DQN in MountainCar

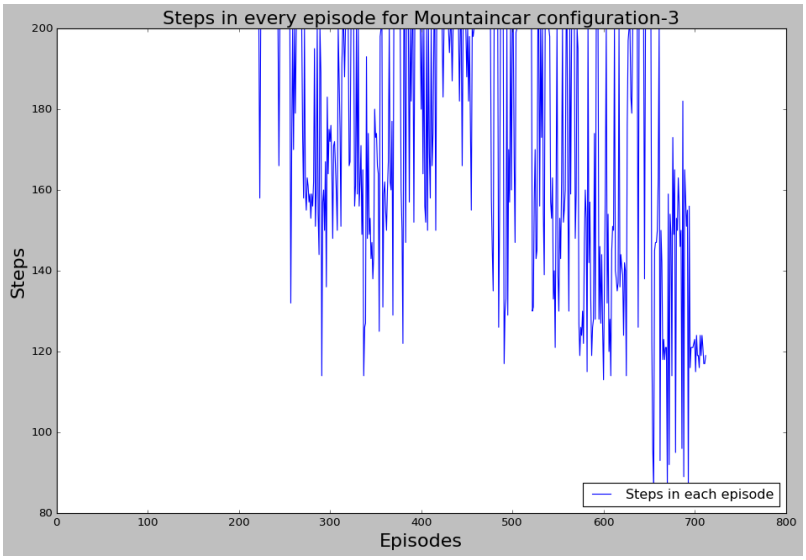

Figure 22: Plot of no. of steps per episode for DQN in MountainCar

```
1
2     def learn(self, experiences):
3         states, actions, rewards, next_states, dones = experiences
4         Q_targets_next = self.qnetwork_target(next_states).detach()
    .max(1)[0].unsqueeze(1)
5
```

```
        Q_targets = rewards + (self.gamma * Q_targets_next * (1 -
    dones))

        Q_expected = self.qnetwork_local(states).gather(1, actions)

        loss = F.mse_loss(Q_expected, Q_targets)

        self.optimizer.zero_grad()
        loss.backward()

        for param in self.qnetwork_local.parameters():
            param.grad.data.clamp_(-1, 1)

        self.optimizer.step()
```

Listing 1: DQN code snippet

The above snippet is used in the DQN implementation for Q-updates. The function learn calculates the $Q_{targets}$ using the target Q-network. This error of the current Q networks's value from the target, is found and backpropagated to update the local Q networks weight. The argument experiences consists of sampled experience from a replay buffer. The target network is updated using a soft update rule.

3 Actor-Critic

3.1 Acrobot

3.1.1 1-step

- Learning rate= 10^{-4}
- NN: 2 hidden layers with 1024, 512 neurons respectively
- No.of episodes to solve: 10

3.1.2 n-step

First configuration:

- n=4
- Learning rate: 10^{-4}
- NN: 2 hidden layers with 1024, 512 neurons respectively
- No.of episodes to solve: 316

Second configuration:

- n=15
- Learning rate: 10^{-4}

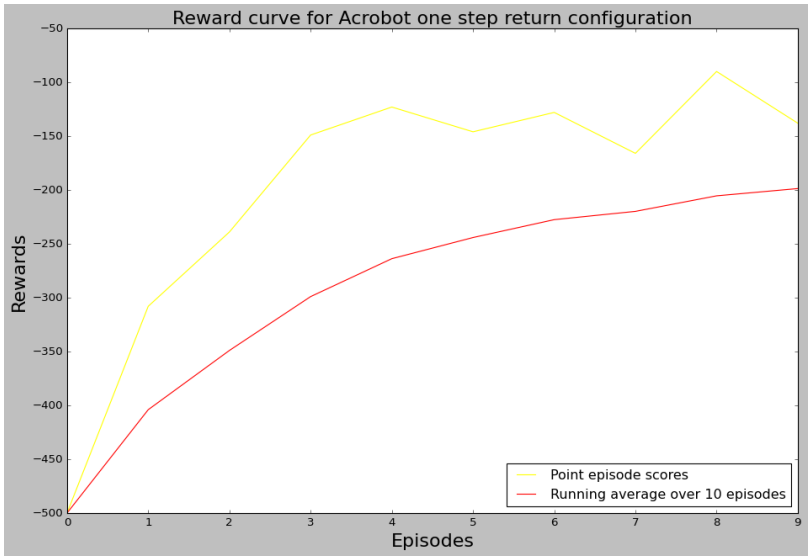

Figure 23: Reward curve for 1-step Actor-critic for Acrobot

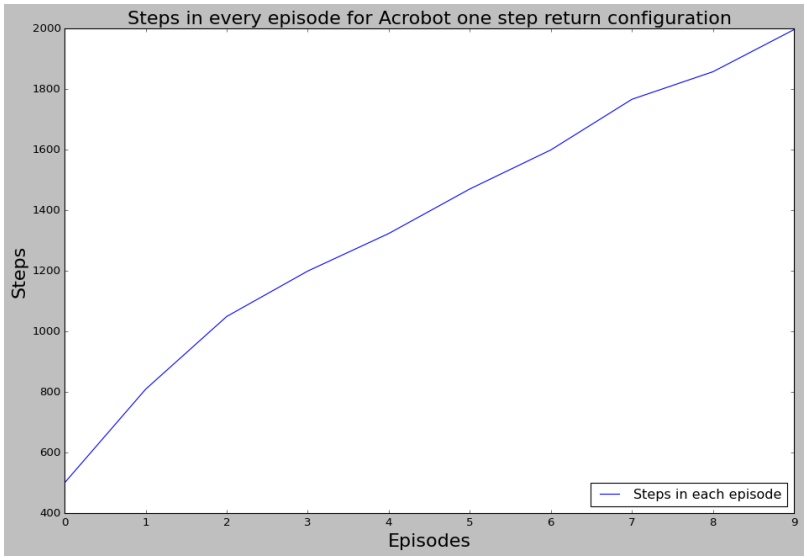

Figure 24: Plot of no. of steps per episode for 1-step Actor-critic for Acrobot

- NN: 2 hidden layers with 1024, 512 neurons respectively
- No. of episodes to solve: 682

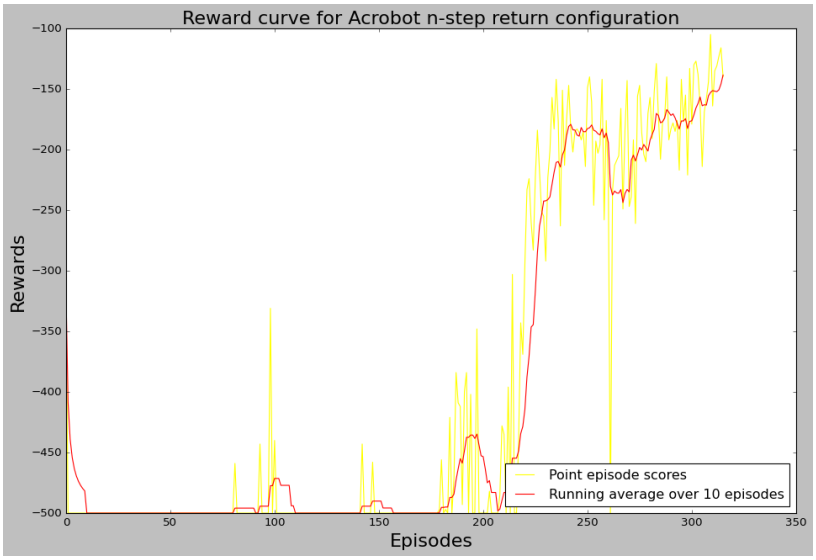

Figure 25: Reward curve for n-step Actor-critic for Acrobot with n=4

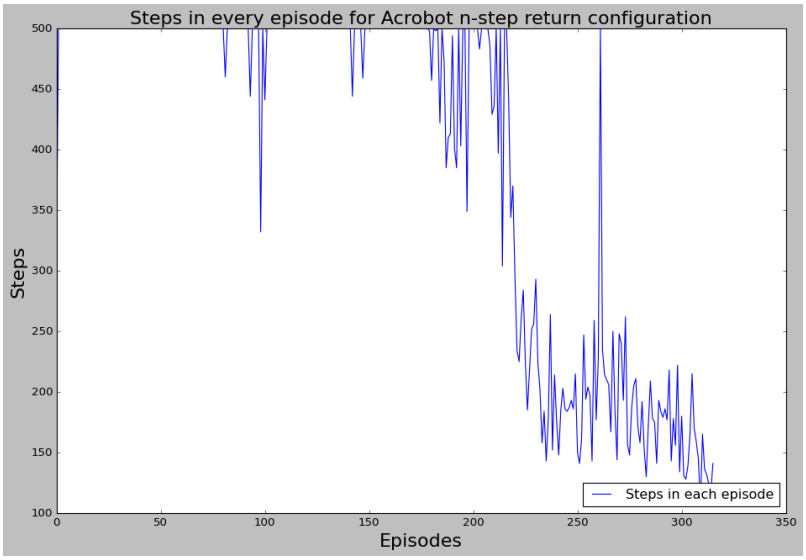

Figure 26: Plot of no. of steps per episode for n-step Actor-critic for Acrobot, with n=4

3.1.3 Full return

- Learning rate: 10^{-3}
- NN: 2 hidden layers with 1024, 512 neurons respectively

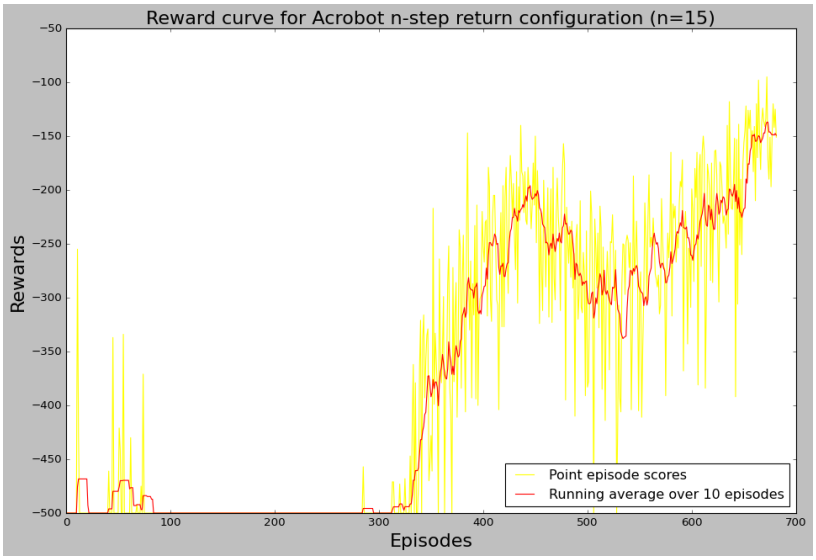

Figure 27: Reward curve for n-step Actor-critic for Acrobot with n=15

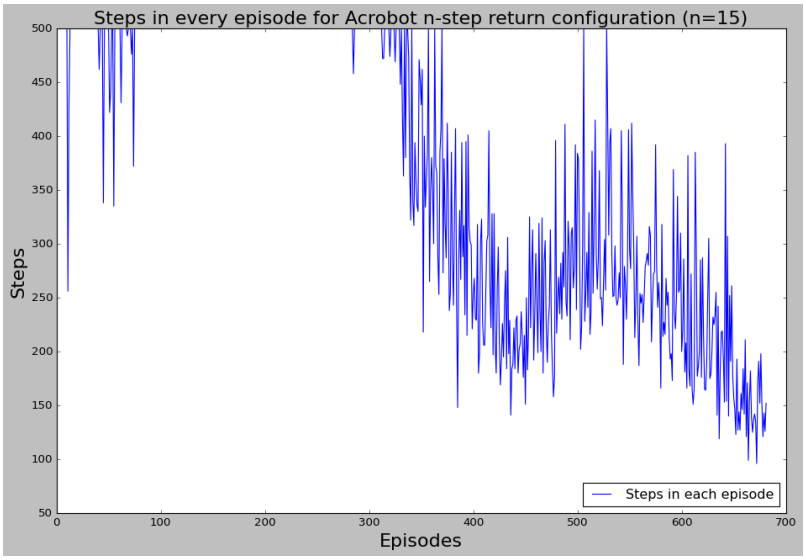

Figure 28: Plot of no. of steps per episode for n-step Actor-critic for Acrobot, with n=15

- No.of episodes to solve: A total of 2500 episodes were used for training, with a duration of 57 mins, but the environment was not solved

Many other configurations were tried, but they were unsuccessful, mainly due

to the high variance in the full-return case

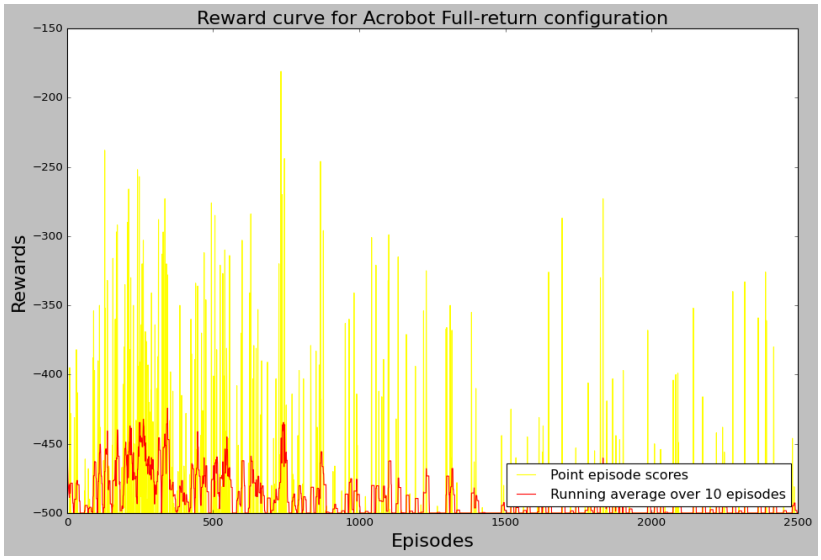

Figure 29: Reward curve for full return Actor-critic for Acrobot

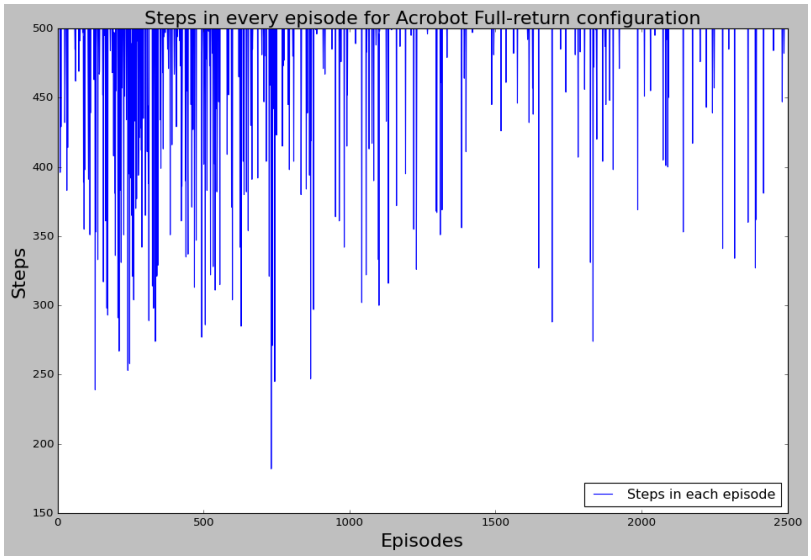

Figure 30: Plot of no. of steps per episode for full return Actor-critic for Acrobot

3.2 CartPole

3.2.1 1-step

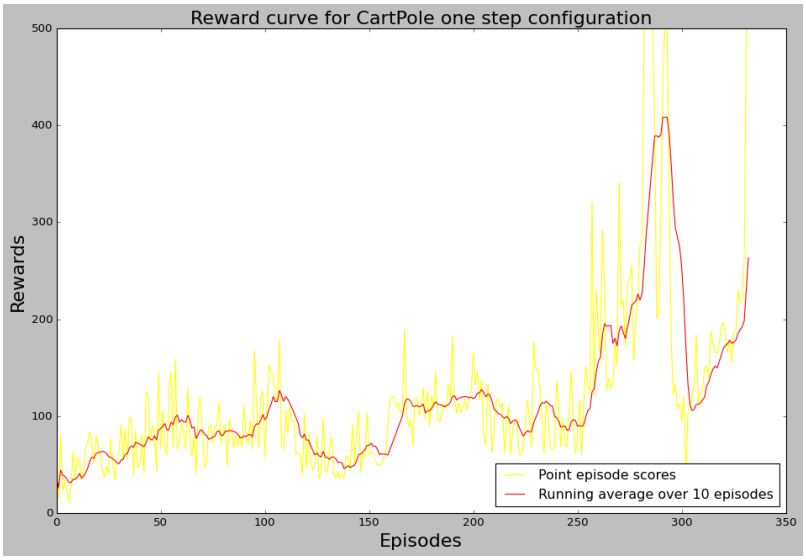

Figure 31: Reward curve for 1-step Actor-critic for Cartpole

- Learning rate= 10^{-4}
- NN: 2 hidden layers with 1024, 512 neurons respectively
- No.of episodes to solve: 333

3.2.2 n-step

First configuration:

- n=20
- Learning rate: 10^{-4}
- NN: 2 hidden layers with 1024, 512 neurons respectively
- No.of episodes to solve: 760

Second configuration:

- n=50
- Learning rate: 10^{-4}
- NN: 2 hidden layers with 1024, 512 neurons respectively
- No. of episodes to solve: 1171

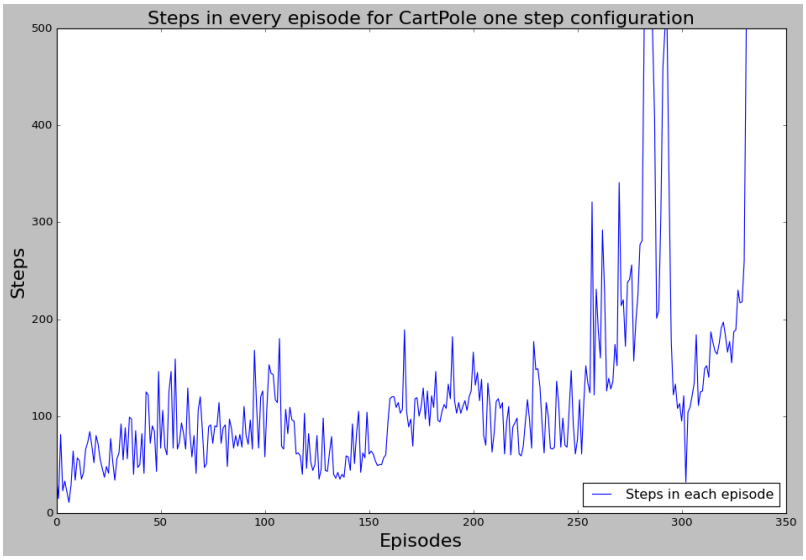

Figure 32: Plot of no. of steps per episode for 1-step Actor-critic for Cartpole

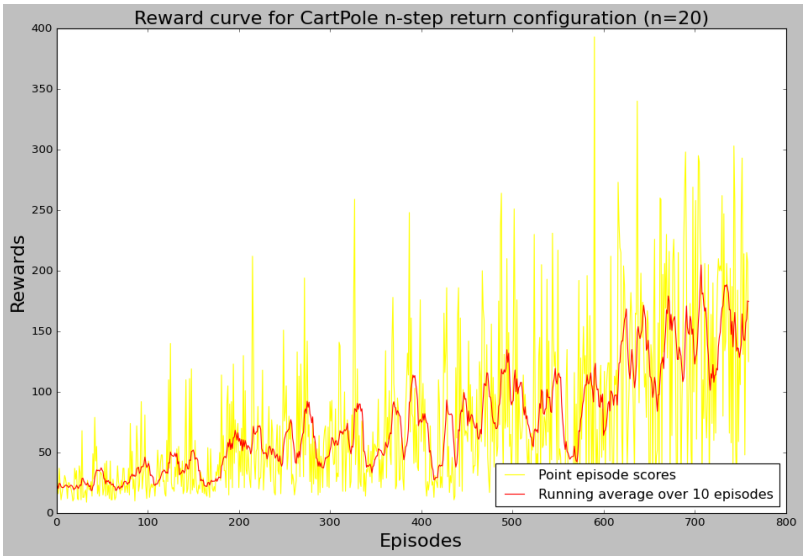

Figure 33: Reward curve for n-step Actor-critic for Cartpole with n=20

3.2.3 Full return

- Learning rate= 10^{-4}
- NN: 2 hidden layers with 1024, 512 neurons respectively

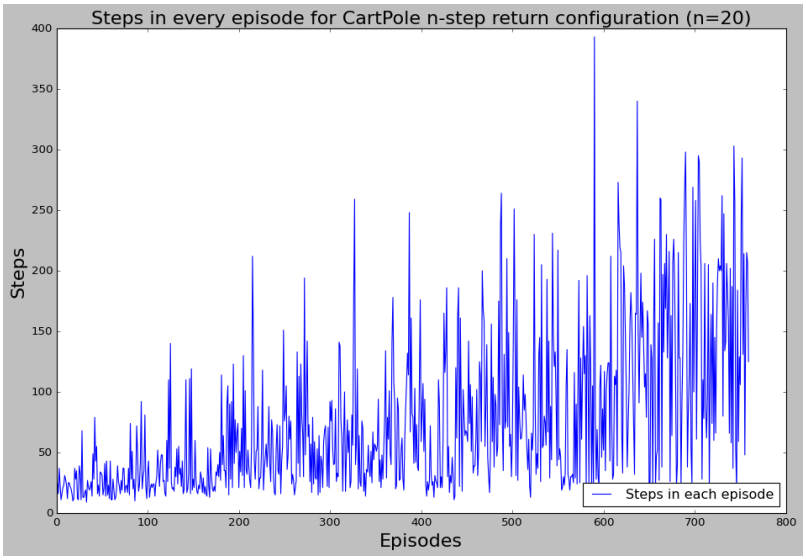

Figure 34: Plot of no. of steps per episode for n-step Actor-critic for Cartpole with n=20

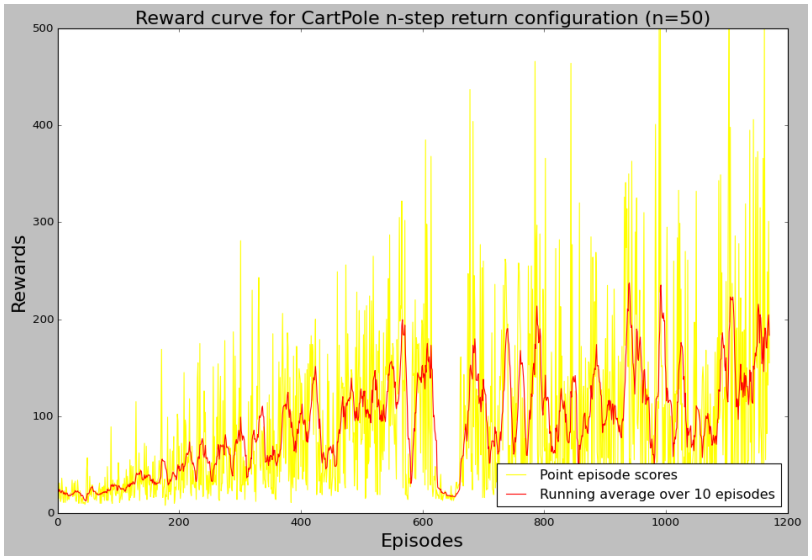

Figure 35: Reward curve for n-step Actor-critic for Cartpole with n=50

- No. of episodes to solve: 1171

For all the three environmnents, multiple configuration were tried by varying

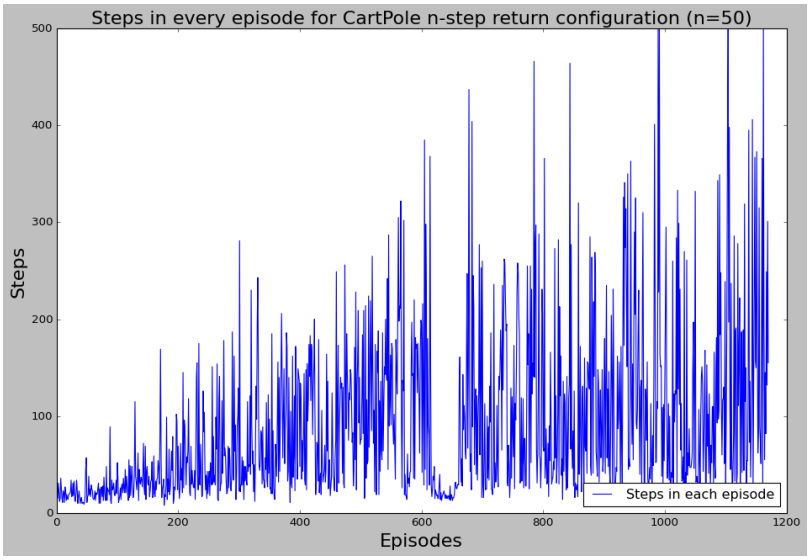

Figure 36: Plot of no. of steps per episode for n-step Actor-critic for Cartpole with n=50

the neural network architecture, but more or less, the same performance was observed. Thus to avoid repetition, they are omitted in this report. All the three variants of Actor Critic performed poorly on the Mountain Car environment, with no significant learning even till 3000 episodes, in many of the configurations tried.

Comparing within 1-step, n-step return and full return Actor Critic, the variance is highest in full return Actor Critic, since it would simply be the Monte Carlo policy gradient. The same is also qualitatively evident from the reward plots. The least variance is in naive 1-step Actor Critic, but has higher bias compared to the n-step and full return versions.

4 Code snippets for Actor-Critic

4.1 1-step

```
    def learn(self, state, action, reward, next_state, done):

        with tf.GradientTape(persistent=True) as tape:
            pi, V_s = self.ac_model(state)
            _, V_s_next = self.ac_model(next_state)

            V_s = tf.squeeze(V_s)
            V_s_next = tf.squeeze(V_s_next)
            delta = reward + self.gamma * V_s_next - V_s
```

```
            loss_a = self.actor_loss(action, pi, delta)
            loss_c =self.critic_loss(delta)
            loss_total = loss_a + loss_c

        gradient = tape.gradient(loss_total, self.ac_model.
    trainable_variables)
        self.ac_model.optimizer.apply_gradients(zip(gradient, self.
    ac_model.trainable_variables))
```

Listing 2: 1-step Actor Critic

For 1-step Actor Critic, the function learn is called for every transition in an episode, and the TD-error delta is computed, after which the critc loss and the actor loss is calculated. The gradients are computed in an online fashion, after the parameters are updated.

4.2 n-step and Full return

```
def learn(self,states,actions,rewards,n,method):

      length=len(states)
      pi_list=[]
      bootstrap_val=[]
      val_state=[]
      rewards=np.concatenate((np.array(rewards),np.zeros(n)))

      if method=="n-step":
        discount_window=[]
        for i in range(n+1):
          discount_window.append(self.gamma**i)
        discount_window=torch.tensor(discount_window)

      for i,state in enumerate(states):
        pi, V_s = self.ac_model(state)
        _, V_s_next = self.ac_model(states[min(i+n,length-1)])
        V_s=V_s.squeeze()
        V_s_next = V_s_next.squeeze()
        pi_list.append(pi)
        val_state.append(V_s)
        bootstrap_val.append(V_s_next)

      if method=='n-step':

        loss_total=0
        for i,next_state_val,state_val in zip(np.arange(length-1)
    ,bootstrap_val[:-1],val_state[:-1]):
          target=torch.cat((torch.from_numpy(rewards[i:i+n]),
    next_state_val.reshape(1))).to(torch.float32)
          delta=torch.dot(discount_window,target)-state_val
          loss_a = self.actor_loss(actions[i], pi_list[i], delta)
          loss_c = self.critic_loss(delta)
          loss_total+= loss_a + loss_c
        self.update(loss_total)

      else:
        loss_total=0
```

```
37          for i,state_val in zip(np.arange(length-1),val_state
    [:-1]):
38            target=(torch.from_numpy(rewards[i:length-1])).to(torch
    .float32)
39            discount_window=[]
40            for j in range(length-1-i):
41              discount_window.append(self.gamma**j)
42            discount_window=torch.tensor(discount_window)
43            delta=torch.dot(discount_window,target)-state_val
44            loss_a = self.actor_loss(actions[i], pi_list[i], delta)
45            loss_c =self.critic_loss(delta)
46            loss_total+= loss_a + loss_c
47          self.update(loss_total)
```

Listing 3: n-step and Full return

For the n-step and full return, the learn function takes as input, the entire trajectory of states, action and rewards in an episode. Firstly, based on whether its n-step or full return, the discounting window is computed, which is an array with discounting coefficients eg: $[1, \gamma, \gamma^2, \cdots, \gamma^n]$. Then, the current value of each state and the current policy given that state, is stored in *val_state* and *pi_list* respectively. The value of states n-step further in the trajectory from the current state is stored in *bootsrap_val*. Then the target for value function of each state is computed by concatenating the rewards until n-steps with the value function of the following state, and then taking its dot product with the *discount_window*. After this, the loss is calculated and accumulated across all states and gradient descent and weight updates, are similar to the 1-step AC.